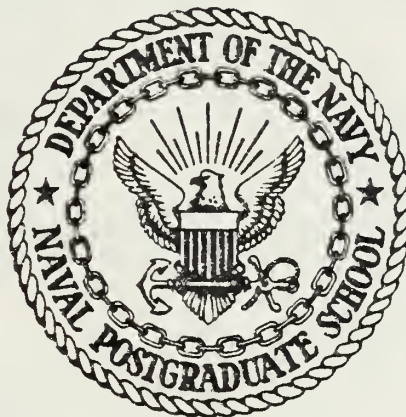


CALIFORNIA 93943

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

VLSI DESIGN OF A VERY FAST PIPELINED
CARRY LOOK AHEAD ADDER

by

Joseph R. Conradi

and

Bruce R. Hauenstein

September 1983

Thesis Advisor:

D. E. Kirk

Approved for public release, distribution unlimited

T215254

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) VLSI Design of a Very Fast Pipelined Carry Look Ahead Adder		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joseph R. Conradi and Bruce R. Hauenstein		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE September 1983
		13. NUMBER OF PAGES 212
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) CAD Tools, VLSI Design, 16-Bit Pipelined Adder		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis is an introduction to the use of computer-aided design (CAD) tools for the design of very large scale integrated circuits (VLSI). The techniques are described and a tutorial is given which illustrates their use in the		

computing environment at the Naval Postgraduate School. The CAD tools were applied to design a 16-bit fast pipelined adder.

Approved for public release, distribution unlimited.

**VLSI Design of A
16 Bit Very Fast Pipelined Carry Look Ahead Adder**

by

Joseph Robert Conradi
Lieutenant, United States Navy
B.S., University of Louisville, 1977
and
Bruce Robert Hauenstein
Lieutenant, United States Navy
B.S., University of Louisville, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1983

thesis

7732

1/

ABSTRACT

This thesis is an introduction to the use of computer-aided design (CAD) tools for the design of very large scale integrated circuits (VLSI). The techniques are described and a tutorial is given which illustrates their use in the computing environment at the Naval Postgraduate School. The CAD tools were applied to design a 16-bit fast pipelined adder.

TABLE OF CONTENTS

I. INTRODUCTION	11
II. OVERVIEW OF VLSI DESIGN	13
A. INTRODUCTION	13
B. VLSI CIRCUITRY	15
1. Basic Transistors	15
2. Basic Gates	17
3. Basic Circuitry	17
C. METHODOLOGY	22
1. Layout	23
2. ILOGS	24
3. Design Rules	24
4. Building Block Approach To VLSI Design	25
5. CAD Tools	25
a. PLA Generator	26
b. CLL-Chip Layout Language	26
c. DRC Design Rule Checker	26
d. Circuit Extractor	26
e. Simulator	26
D. FABRICATION	27
1. File Generation	27
2. MOSIS	27
3. Pattern Generator and Maskmaking	27

4. Patterning	28
5. Packaging	28
E. TESTING	28
F. SUMMARY	29
III. VLSI CAD TOOLS	30
A. LOGIN PROFILE	30
B. FUNCTION OF SOURCE PROGRAMS	32
1. Chip Layout Language (CLL)	32
a. Cif	34
b. Cifload	34
c. Merge	34
d. Rplot	34
e. Rsort	35
f. Tplot	35
g. Window	35
2. Supporting Programs	35
a. Cifar	35
b. Convert	36
c. Plagen	36
d. Plague	36
e. Unconvert	36
3. Design Rule Checker (DRC)	36
4. Circuit Extractor	37
a. Extract	38
b. Node-plot	38

c. Sim	38
5. Static Checker	38
6. Event Level Simulator	39
IV. GENERATING CIF USING CLL	40
A. TUTORIAL	40
1. Header	41
a. Comments	41
b. External Symbols	41
c. Defines	42
d. Includes	42
e. Conditionals	42
2. Symbol Definition	43
3. Body	43
a. Comments	44
b. Rectangles	44
c. Layers	44
d. Wires	45
e. Vias	46
f. Calls	46
g. Iteration	47
h. Expressions	47
i. Print	48
B. CELL LIBRARY	48
C. USING CLL	48
1. Making Files	49

2. Plotting	49
3. Creating CIF	50
D. EXAMPLE	51
V. DESIGN VALIDATION	58
A. DESIGN RULE CHECKER	58
1. Evaluation Of Outputs	59
2. Example	59
B. CIRCUIT EXTRACTOR	64
1. Plotting	64
2. Defining Nodes	65
3. Creating A Simulation File	65
4. Example	65
C. STATIC CHECKER	68
1. Evaluation Of Outputs	68
2. Example	69
D. EVENT SIMULATOR	70
1. Using Esim	71
2. Example	71
VI. PROJECT: 16 BIT VERY FAST PIPELINED	
CARRY LOOK AHEAD ADDER	74
A. INTRODUCTION	74
B. LOGIC DESIGN	74
1. Pipelining	74
2. Carry-Look-Ahead Addition	76
3. Design Considerations	83

C. DESIGN VERIFICATION	86
D. LAYOUT	97
E. DRC	103
F. SIMULATION	103
VII TESTING	105
A. EXPECTATIONS	105
B. PROCEDURES	110
C. RESULTS	112
VIII. CONCLUSION	116
A. SUMMARY	116
B. RECOMMENDATIONS	116
APPENDIX A - INTRODUCTION TO THE VAX-11/780 AND UNIX	118
APPENDIX B - MANUAL PAGES FOR VLSI CAD TOOLS	141
APPENDIX C - SUMMARY OF CLL COMMANDS	176
APPENDIX D - DESIGN FABRICATION	180
APPENDIX E - FILES AND PROGRAMS FOR THESIS PROJECT	188
LIST OF REFERENCES	210
INITIAL DISTRIBUTION LIST	211

ACKNOWLEDGEMENTS

We would like to thank the following individuals for their assistance in the completion of this thesis:

Naval Postgraduate School

Dr. Donald Kirk

Prof. Robert Strum

Dr. Herschel Loomis

Mr. Al Wong

Stanford University

Dr. Robert Mathews

Ms. Susan Taylor

Ms. Irene Watson

Mr. Ernest Wood

Air Force Institute of Technology

Lt.Col. Harold Carter

I. INTRODUCTION

Advances in computer-aided design (CAD) and fabrication techniques, along with the text **Introduction To VLSI Systems** by **Mead and Conway** [REF.1], have created the ability for systems engineers to **custom design** digital integrated circuits. Until recently, the design of integrated circuits has been traditionally carried out by a select group of logic designers working in semiconductor laboratories. Systems engineers had to "make do" or "fit in" the products of these labs to realize their designs. The systems engineers had little participation in the actual design of the chip. The **MEAD and CONWAY** design methodology and computer aided design tools (CAD) have bridged the gap between the systems engineer and the circuit designer. Now, systems engineers can create a custom design to support specific needs. Armed only with a knowledge of circuit and logic design, the present-day chip designer utilizes powerful CAD tools to manipulate basic digital circuits (cells) from a pre-established library in order to realize a custom design. Additional CAD tools can be used to check, evaluate and simulate the design. This thesis, along with minimal references to outside sources, provides a reader who has a basic knowledge of logic design with enough information to design a custom digital integrated circuit of moderate complexity.

Before entering the realm of Very Large Scale Integration (VLSI), a few preliminaries must be covered. "VLSI" as used in this thesis should not be confused with the Very High Speed Integrated Circuit (VHSIC) program in the Department of Defense. This program with a \$400 million budget is charged with advancing the state of the art for the number of devices on a single piece of silicon, operating speed, submicrometer line width, and other attributes. Present day commercial VLSI chips are capable of about 130,000 transistors with a

typical number of about 20,000. VHSIC on the other hand has set a goal to produce a circuit containing approximately one million transistors per integrated circuit by the end of the decade. This thesis deals with devices of moderate complexity, that is, from a few gates up to the size of small commercial products. A typical number would be on the order of 2,000 to 10,000 transistors. Thus, the complexity of the devices considered here is much less than that of commercial and research programs.

This thesis provides an introduction to VLSI circuitry and procedures, CAD software resources and their uses, the VAX 11-780 computer (UNIX operating system) and other hardware resources available at the Naval Postgraduate School. In addition, the creation of a **16 BIT VERY FAST PIPELINED CARRY LOOK AHEAD ADDER** is traced from conception through the design methodology to fabrication and testing. This provides a concrete example so that the interaction between the user, the software and the hardware may be more fully understood.

II. OVERVIEW OF VLSI DESIGN

A. INTRODUCTION

The design methodology in this thesis applies to "digital" systems-- "analog" systems are not considered. Digital systems inherently use highly regular and repetitive structures. Many digital devices have data paths sixteen bits wide, and path widths of thirty-two to sixty bits are not uncommon. Memory units, arithmetic logic units (ALU's), shift registers, crossbar switches, etc. all possess uniform repetitive structures. Combinatorial control logic in many cases can be realized using programmable logic arrays (PLA's) which are also "highly structured". In addition, digital systems operate using a high or low voltage to represent one or the other of two binary states. The two preceding attributes of digital systems are not prominent in most analog circuitry and therefore analog devices do "not" readily lend themselves to the design methodology described here.

Because digital systems are highly repetitive, highly structured and operate in either the "on" or "off" fashion, they can be realized by using the simplest of logic gates. When these simple logic gates are fabricated in silicon, they form a very regular array of rectangles strategically scaled and properly placed. Even the interconnecting "wire" runs are rectangles with one dimension (length) much larger than the other dimension (width). Resistors are realized by the predictable resistance of a "depletion mode" metallic oxide semiconductor field effect transistor (MOSFET) whose gate region is connected to its source. Microscopic inspection of a high density integrated digital circuit would reveal only squares or rectangles of varying dimensions and heights. The variation in height of these elementary figures results from the placement of layers of conducting materials onto the surface of the chip.

Integrated systems in nMOS technology contain three levels of conducting material separated by intervening layers of silicon dioxide (insulating material). They are from top to bottom: **metal, polysilicon, and diffusion**. All three paths conduct electricity well enough to be considered wires. Unless the layers are specifically intended to be electrically connected by using contact cuts, paths on the metal layer have no significant effect on the "poly" or the diffusion layer. But, when a path on a poly layer crosses over a path on the diffusion layer an "enhancement" mode MOSFET is formed. This transistor is effectively an electronic switch. Various forms and interconnections of this electronic switch provide the basic building blocks from which large scale systems are designed.

The n-channel MOS process is by far the most mature process in the field of VLSI. Most devices now produced use nMOS processes, but there are also other processes. For example, pMOS stands for p-channel MOS (the "p" denotes positive type carriers in the channel beneath the gate area as "n" signifies negative type carriers). CMOS denotes complementary MOS which utilizes a combination of the two for individual devices. And "mixed" MOS utilizes "n" and "p" MOS at different locations on the device. CMOS-SOS is CMOS but is formed on a sapphire surface to increase the operating speed (SOS signifies silicon on sapphire). Bipolar transistor architecture also has a place in VLSI. Since the nMOS process is the most established, and because the project created in this thesis is of the nMOS type, we shall concentrate on it. This should not imply that nMOS is the best method. Other processes may be better in terms of power consumption, speed, device density etc. However, complexity in the actual fabrication and design may outweigh some of these more desirable traits.

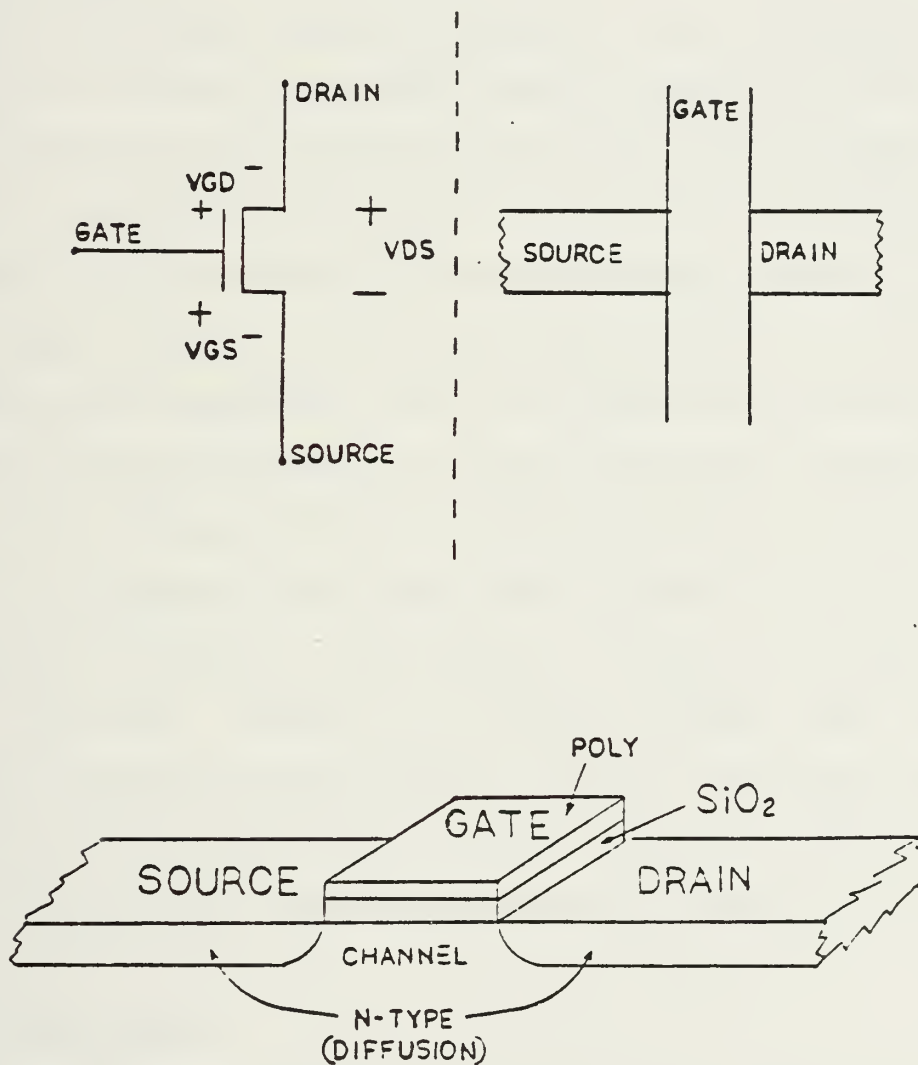
B. VLSI CIRCUITRY

Mead and Conway [REF.1] provides an excellent discussion in chapter one concerning the basic devices and circuits needed to understand and solve typically encountered systems problems. A full and complete understanding of this chapter, however, is not a necessity to be able to design a custom chip. Most of the devices and circuits discussed in chapter one of [REF.1] will be presented in the following discussion which should provide the depth necessary to continue and successfully complete a custom design.

1. Basic Transistors

The nMOS transistor is the most basic device used in VLSI circuitry. Shown in Figure(2.1) are three different representations of the same device.

A positive voltage on the gate of an nMOS transistor is used to control the movement of negative charges between the source and drain. When the voltage on the gate enhances the quantity of negative charge carriers(electrons) under the gate in order that current may flow between source and drain, the device is labeled an enhancement mode transistor. The enhancement mode transistor by itself is effectively a switch and is referred to as a "pass" transistor. When a positive voltage is applied to the gate, the switch is closed. When a voltage below a certain threshold is applied,the switch is open. When the area under the gate region of a transistor already has enough negative charge carriers to support current flow between the source and drain with no voltage applied to the gate,the device is called a depletion mode transistor. The excess supply of charge carriers is supplied by a doping process during fabrication. The area of excess carriers is called the ion implant region. The depletion mode transistor is always on unless a voltage of proper polarity(negative for nMOS) is applied to the gate to deplete the number of charge carriers, thereby turning off the switch. In the enhancement mode device the region under the gate area must be enhanced



Figure(2.1) The Basic nMOS Transistor

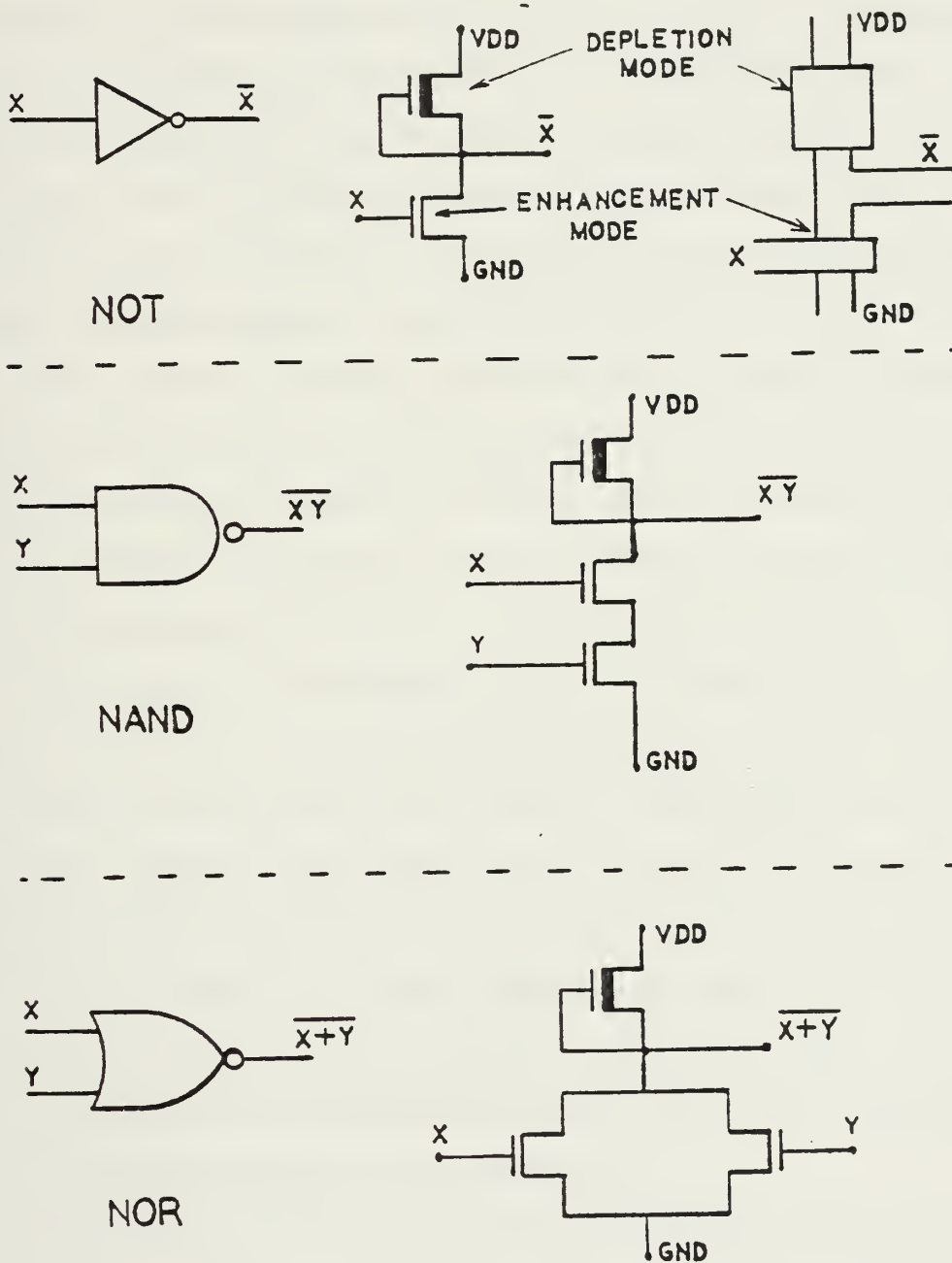
to turn the switch on while in the depletion mode device the region under the gate area must be depleted to turn the device off. In a pMOS device, the operation is identical except that the charge carriers are "holes" and voltage polarities are just the opposite of that required for proper nMOS operation.

2. Basic Gates

The basic inverter will now be discussed. Using an enhancement mode switch (pass transistor) in series with a resistor, an inverter gate can be realized. In VLSI design, however, resistors are not used. Instead, resistance is generated by a depletion mode transistor. To ensure that the depletion mode transistor remains in the "on" mode, thereby effectively introducing a predictable amount of resistance as the load, the gate is connected to the source. A resistance made of polysilicon or carbon would take up far too much area on the surface of the chip to allow reasonable densities. The amount of resistance introduced by this continually switched on transistor is largely determined by the size of the gate and ion implant region. More important is the ratio of the gate geometries of the depletion mode ("pull-up") transistor to the enhancement mode ("pull-down") transistor. By obtaining proper ratios for the depletion/enhancement mode transistors, an inverter circuit can be produced. The output will be the complement of the input. Shown in Figure(2.2) is the basic inverter in several forms along with extensions that realize the NAND and NOR functions.

3. Basic Circuitry

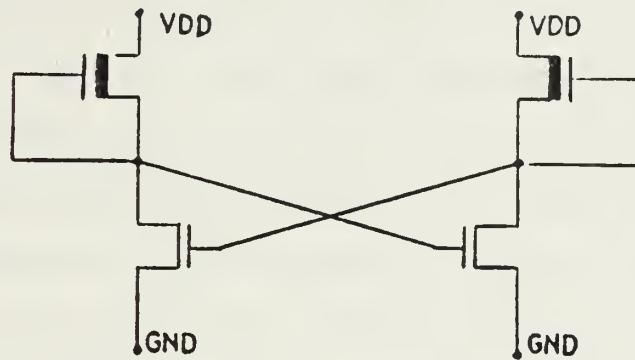
Many applications require that the output of a basic inverter drive more than one following circuit(fanout). In this case, because of the much larger combined input capacitance, more drive current capability is required. Again, manipulation of the basic inverter produces both inverting and non-inverting "super buffers". These are high performance circuits used to reduce the delay time



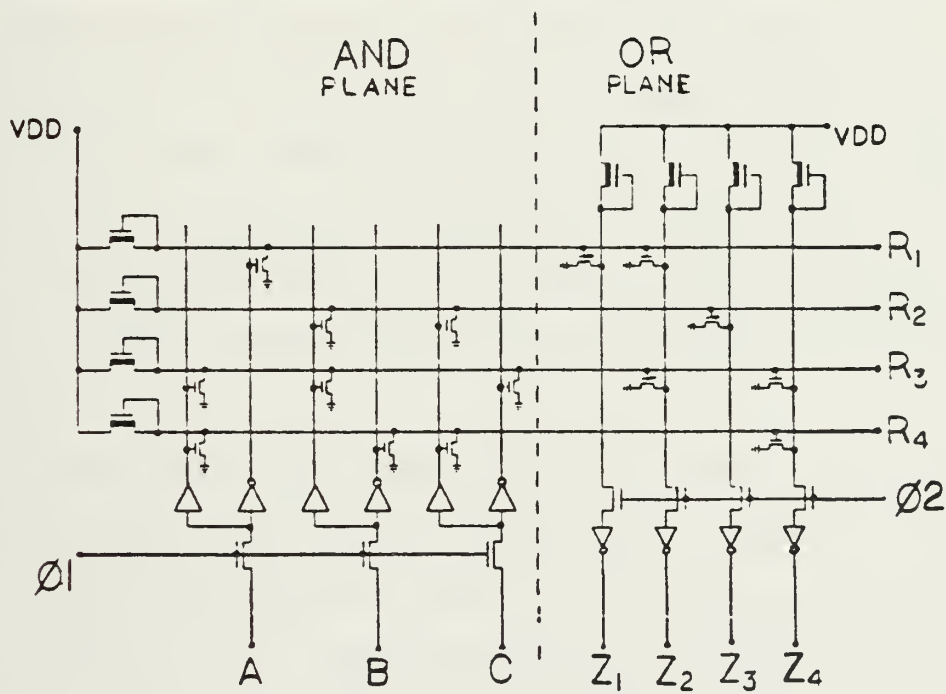
Figure(2.2) The Basic Inverter with NAND and NOR extensions.

that is induced by the increased RC time constant when fanout and parasitic capacitances cause the equivalent capacitance to increase. The extra drive current capability is obtained from proper interconnection of two standard inverter gates. See [REF 1], Figure 1.21 and 1.22, for a schematic representation of inverting and non-inverting super buffers. To emphasize that nearly all circuits can be constructed through the proper connection and adaptation of the basic inverter gate, a few additional examples will be discussed. Shown in Figure(2.3) is the cross-coupled inverter circuit. This circuit has many applications in control sequencing, memory cells, and register arrays. A programmable logic array (PLA) is shown in Figure(2.4). Normally, PLA's are thought of as having an AND plane and an OR plane. Careful analysis shows that the PLA is made up of nothing more than pass transistors and inverter gates. Actually, this PLA implements the NOR-NOR canonical form of Boolean functions of the inputs. By properly feeding selected outputs back into selected inputs, a synchronous finite state machine results. PLA's prove to be very important in system control sequences. One of the CAD tools which is discussed in a following chapter is called PLAGUE, which stands for PLA Generator Using Equations. By inserting the Boolean equations in the proper format, the software tool determines the proper placement of the elementary figures(rectangles) to realize the desired logic in silicon. PLAGUE can realize combinatorial logic on the order of 40 inputs, 40 outputs and 150 product terms. [REF.1] provides excellent information on PLA's and their uses. The design project which is the subject of this thesis (a sixteen-bit adder) relies heavily on the use of PLA's.

Referring to Figure(2.4), an implication arises when observing the input and output "registers". Clearly, if the input and the output registers are made up of nothing more than pass transistors and inverter gates, then to truly be a register, some type of storage mechanism must be involved. This is indeed true.



Figure(2.3) Cross-coupled Inverter Circuit

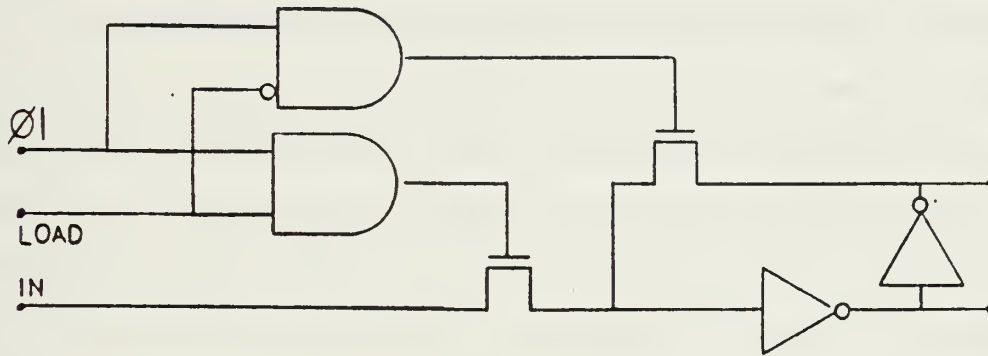


Figure(2.4) Inverter Realization of PLA

When a positive voltage is applied through an "on" pass transistor to the gate of the inverter circuits, the capacitance between the gate and the substrate is charged and maintains the charged condition for a finite amount of time after the pass transistor is turned off. The "turned off" pass transistor represents a large amount of resistance. This charge will decrease in an RC time constant fashion. The amount of time for the gate voltage to decrease to below threshold is on the order of milliseconds. Threshold voltage is that value of voltage necessary to be considered a "high" voltage thereby causing the output of the inverter to appear as a "low" voltage. Thus, for proper operation, the dynamic registers must have their inputs updated and outputs utilized at a clock period less than this "bleed-off" time of the charge stored on the gates. For this type of PLA input/output register scheme, the clock period cannot be too low, or erroneous results may be obtained. The upper clock frequency is limited by the amount of time it takes for the basic inverters in the NOR planes to switch to the proper output voltages once the input voltages and clock pulse are applied. There is a detrimental effect when several inverters are cascaded in series as well as in parallel (fanout) - the voltage must be given time to ripple through all levels of logic. The time it takes to charge up the additional parasitic capacitances and logic gates to realize the proper output is the limiting factor for the maximum clock rate.

To overcome this effect of charge bleed off, an inventive "refresh" scheme is utilized in the selectively loadable dynamic register cell shown in Figure(2.5). Using the control signals LOAD and phase 1 of the system clock, this scheme allows the register cell to be selectively loaded and "refreshed". This alleviates the problem of the voltage dropping to below threshold. This circuit may be used to solve many of the storage applications needed in VLSI systems.

Thus, nearly all functions needed to realize a digital system can be obtained by manipulation of the basic inverter circuit and pass transistor. The next step is to become familiar with the design methodology.



Figure(2.5) Loadable Register Cell

C. METHODOLOGY

There are several reasons for developing VLSI digital systems. A new need may force the creation of a custom designed system. It may be required, or desired, to condense the size of existing designs, usually in the form of printed circuit board systems, for other applications. Also, improvements in VLSI technology may allow already functional chips to be made smaller, thereby allowing more functional units to be placed on a single chip. For whatever reason a system is developed, the design usually begins in the mind of an engineer. Existing functional units such as shift registers, memories, ALU's, PLA's, flip-flops, etc. normally provide the building blocks for the design. New functional units, along with unique methods of interconnection, usually appear in a more "skeletal" form to clearly define the unit's purpose. The CAD resources available

to the engineer determine where the pencil and paper approach to the design can be replaced by ever improving CAD tools.

1. Layout

Since VLSI designs deal almost uniquely with inverters and pass transistors, it is not necessary to initiate the design at the schematic level. Rather, the skeletal form that is mostly used is called the "stick" layout or design method. The stick method involves the color coding of the different conducting materials used on the chip. Green is used for diffusion. Red is used for polysilicon. Yellow is for the ion implant region. Blue stands for the metal layer. Black represents a contact cut. In some cases logic symbols are also used in the stick diagram. This skeletal form is known as mixed notation. For good color examples of the stick and mixed notation and the corresponding geometric layout refer to [REF.1] color plates 4 through 8. It should be evident from these color plates that wherever a red poly path crosses a green diffusion path an MOS transistor is formed. Similarly, where red crosses over green which in turn crosses over yellow, a depletion mode transistor is formed when the gate region is connected to the source. The stick methods was mainly developed for hand layout. However, recent advances in CAD tools and color graphics terminals, allow the stick method to be readily adapted to computer design thereby alleviating the pencil and paper approach. Using these sophisticated design tools, stick diagrams can be drawn and manipulated directly on the color terminal to realize the design. When the designer is satisfied with his creation, the stick diagram can be automatically compacted (observing critical design rules) and mapped into the geometric rectangular layout necessary for fabrication. Designers not having access to this level of sophistication must rely on a balance between the pencil and paper approach and CAD at this stage of the design.

2. ILOGS

CAD tools exist that provide a valuable link between the pencil and paper approach and the CAD portion of the design. These tools are not necessarily related to VLSI design. ILOGS is an Interactive LOGic Simulator. Before "charging" into the realm of VLSI layout, circuit extraction, stipple plots, refinement of the layout and simulation, it is wise to emulate the design using ILOGS, or a similar tool, to verify functional correctness. For smaller designs, or when the software resources are highly sophisticated, this design verification step may not be absolutely necessary. Nevertheless, successful emulation of the design invokes a sense of confidence in the designer. It is highly recommended that the first attempt at custom VLSI be initiated with a verification of the design using ILOGS. The project (16 BIT ADDER) was initially verified using this method. A description on how to use the ILOGS program is discussed in chapter six.

3. Design Rules

A key point in VLSI design methodology will now be discussed. Design rules are layout rules that result from analysis of semiconductor physics and fabrication processes. It is not necessary for the systems engineer to be thoroughly cognizant of how the rules were developed. It is necessary, however, for the designer to know what the rules are and to abide by them. Design rules are geometric constraints placed on the basic rectangles concerning minimum allowable separations, extensions, widths, and overlaps in the various levels of the chip. Since various processes in creating VLSI chips are improving and ever decreasing in feature sizes, it has become convenient to develop these rule in terms of a "length unit" denoted as (λ) lambda. Present day nMOS processes are typically 2.5 microns (μm). Another way to look at this length unit in this case is $\lambda = 2.5\mu\text{m}$. When using a 2.5 micron process, for example, the minimum distance allowed between two "wire" runs of poly is 2 times lambda or 5.0 microns.

However, when the process is improved/decreased to $\lambda = 1.25$ microns, the rule of 2 times λ separation still applies but now the actual distance is 2.5 microns. This results from the fact that every dimension on the chip has also been scaled down accordingly. See [REF.1] color plates 2 and 3 for an excellent description of the basic design rules.

4. Building Block Approach To VLSI Design

A VLSI system can be visualized as a large complex puzzle with the pieces located in a box called the cell library. The cell library consists of pre-designed, pre-tested cells in geometric forms that have been created by VLSI design engineers. Some of the cells may be very basic while others may be quite complex. For an excellent description of several cells contained in most libraries, refer to color plates 9 through 15 of [REF.1]. Plate 9 shows the correlation between the stick diagram and actual hand layout of a basic shift register cell. The task for the systems engineer in order to realize the custom design is to select, adapt, replicate, manipulate, and orient the proper cells to form functional units. These functional units are then properly positioned and interconnected to each other and to the outside world (through the use of bonding pads) to complete the puzzle. It is this building block approach to VLSI design that provides the strongest connection between the chip designer and the systems engineer. It is assumed that the reader has access to a cell library as well as the necessary CAD tools before attempting a VLSI custom design. The exploration of the design and construction of the cell library is beyond the scope of this thesis. Here we are oriented towards the use of a cell library with assorted CAD tools.

5. CAD Tools

Chapters three, four and five are devoted to VLSI-CAD tools. However, for continuity, a basic explanation of several of the basic tools is provided in the following sections.

a. PLA Generator

The purpose of the PLA generator when used in conjunction with the PLAGUE software is to create a PLA cell that can be added to the existing cell library. This PLA cell can then be manipulated, adapted, oriented etc. as any other of the library cells.

b. CLL-Chip Layout Language

CLL is the software tool that provides the capability for the manipulation, replication, adaptation, orientation, and placement of the various cells. It also provides a means of interconnecting the functional units with each other and to the outside world through the use of the "wire-list" commands.

c. DRC Design Rule Checker

The final design is scrutinized by the design rule checker. It will make known to the designer if and where any of the design rules are violated. Even at mini-computer speeds, this program's execution time is rather lengthy.

d. Circuit Extractor

The circuit extractor is used to define nodes in the design in order to perform a functional test or simulation.

e. Simulator

The simulator uses node definitions obtained in the circuit extractor portion and processes information received from the designer. The designer inputs information and looks for expected results. In the case of the thesis project, two 16-bit vectors consisting of 1's or 0's are used as an input and the sum of the two input vectors is expected at the output nodes (provided there are no errors in the circuit).

The above tools may carry different names and exist at different levels of sophistication, but they represent a reasonable cross section of the available VLSI design tools. CAD tools will be discussed in detail in later sections.

D. FABRICATION

Upon completion of a successful design rule check and a correct simulation of the design, it is reasonably safe to assume that the design is ready to be fabricated. To this point, nothing has been mentioned about how the design arrives at the implementation service, the form in which it is sent, or what events take place after the design is delivered.

1. File Generation

One of the products of the chip layout language tool is the Caltech Intermediate Form file (CIF file). The CIF file is a standard machine readable form for representing integrated system layouts. Its purpose is to unambiguously describe the dimensions and layer of each geometric figure (rectangle) to a pattern generator.

2. MOSIS

MOSIS is an acronym for Metal Oxide Semiconductor Implementation Service. This is the institution that receives the design in the Caltech Intermediate Form. The standard means for communication between MOSIS and the designer is the ARPANET (Advanced Research Projects Agency Network). The CIF file is transmitted directly from one computer to the other over standard telephone lines. The implementation service, after several preliminary checks, forwards the CIF file to a maskmaking company.

3. Pattern Generator and Maskmaking

The pattern generator is a very sophisticated computer driven photolithographic device that accepts the CIF file as an input. The pattern generator converts the CIF file to a Pattern Generator file (PG file). The PG files are then used to create the masks through a very delicate "flashing" operation. This flashing operation causes the positions and the dimensions of each layer of rectangles to be imprinted on photo-sensitive material. This material is developed and

then reduced in size. The reduced "negative" is replicated many times in a step-and-repeat fashion in order to produce a template of many identical designs of individual layers of rectangles. The individual designs lie abutted to each other in a side-by-side and top-to-bottom configuration. This template is used to develop the "working" masks, which are then utilized in the fabrication process to pattern the design into the silicon wafer.

4. Patterning

The working masks selectively allow an intense source of radiation, in the form of ultraviolet light, electron beam (E-beam), or low energy X-rays to impinge upon the appropriate layer of the chip surface. This selective exposure to radiation causes a chemical reaction in an organic material, called "resist", previously coated onto the chip surface. The exposed resist can easily be removed while the unexposed resist cannot. After removal, acid etching is performed to pattern the design into the silicon. The nMOS process requires approximately forty-four steps to complete the finished chips.

5. Packaging

The final step before mailing the completed chips back to the designer is packaging. The wafer is diced into individual chips. Each chip is cemented into a package. The bonding pads are connected to very fine wires which in turn are connected to the package leads. A top cover is then bonded over the chip. The completed design is returned to the customer. The time period from CIF file submission to chip receipt is normally three to six weeks.

E. TESTING

There are basically two types of testing that can be done and they depend largely on the resources available to the designer and the complexity of the design. Commercial testers are available. They are very thorough, but

expensive. A company in the business of VLSI design may very well benefit from such a tester. They not only can test various chips for proper operation, but can also aid in the location of a design/fabrication error if one should be present. A custom made testbed is sufficient for many applications, provided that the design is not too complex. The design of a custom testbed, however, could easily become more expensive and time consuming than the actual VLSI design.

F. SUMMARY

This chapter has provided a brief, but rather complete, overview of all aspects of VLSI design. The remaining chapters and appendices provide detailed information on specific software tools, hardware resources, and custom VLSI design methodology. The step-by-step approach utilized to a great extent in the remainder of this thesis should provide the reader with enough information to embark on a custom design.

III. VLSI CAD TOOLS

Prior to attempting a VLSI design using the NPS CAD tools, the designer must have a "working" understanding of the UNIX operating system and *c-shell* commands. If this is not the case, he should read Appendix A and complete the included tutorials.

Various source programs comprise the CAD tools which are used to complete a VLSI design. These programs ensure that the output file is in the proper format and that the chip will be successfully manufactured. In general, these programs work most effectively when used in the prescribed order. (See Figure 3.1 for a flow diagram of the design process.) The project is first conceived using a "top-down" approach (i.e., the overall project is conceived and then broken into lower levels for individual design). Then, if the project is designed using a "bottom-up" approach (i.e., if the lowest level cells and functional units are designed and checked prior to forming the total design), the task will proceed more easily and with less time involved.

In order to aid the designer in utilizing the CAD tools effectively, a functional description of the source programs is provided in this chapter. Careful attention to the following paragraphs will allow the designer to understand and use the VLSI CAD tools.

A. LOGIN PROFILE

Each user of the VAX computer has a standard **login profile** (executive) program which is established by the CS Department. This program is executed each time the user logs into the system and it controls the functions of the terminal. Although this profile is sufficient for using the system commands, the

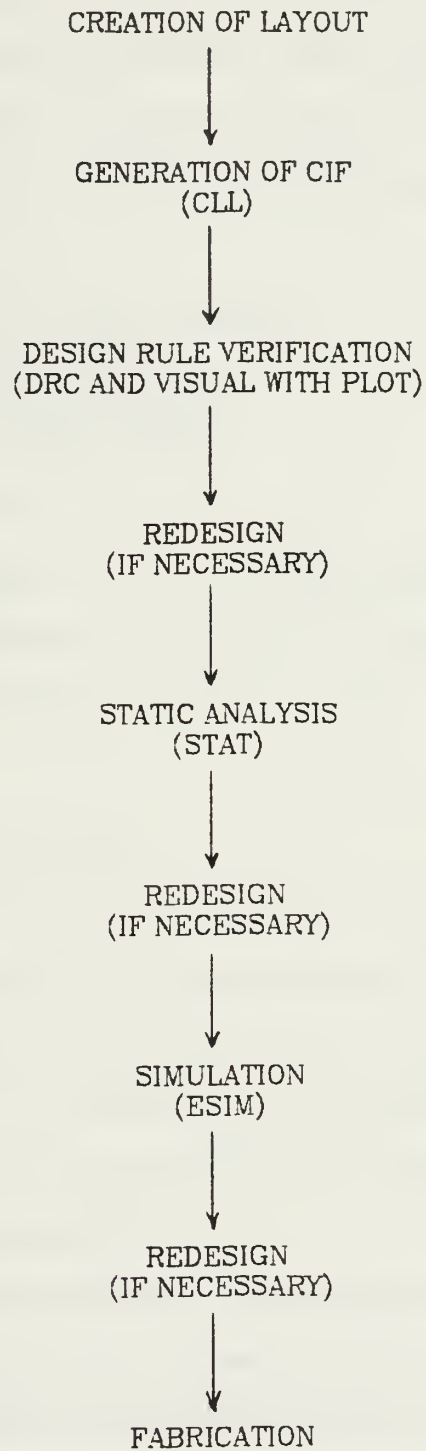


Figure 3.1. VLSI DESIGN PROCESS

designer should obtain, by the steps described below, the standard **VLSI profile** in order to use the CAD tools.

The **login profile** is stored in the user's "login" directory under the name of *.login* and is not listed with a normal **ls** command. (It can be listed with a **ls -a** command which lists all files in a directory.) The **VLSI profile** can be obtained by performing the following steps:

- 1) Change directory to /vlsi/lib/local/work
cd /vlsi/lib/local/work
- 2) Copy *.login* to the "login" directory
cp .login /work/(user-name)
- 3) Change directory to "login" directory
cd
- 4) Log off then log back on.

Now, each time the designer logs onto the UNIX system, he will be able to use the VLSI CAD tools. Additionally, the DEL key will delete previously typed characters (similar to <CTRL>H) and a "period" (.) followed by RETURN can be used to logoff.

B. FUNCTION OF SOURCE PROGRAMS

The source programs used for VLSI design allow the user to generate a Caltech Intermediate Format (CIF) file, check the file for design rule errors, statically check the circuit, and simulate the design to ensure correct implementation. The time involved to complete each of these steps depends both on the complexity of the design file and on the actual tool being used.

1. Chip Layout Language (CLL)

The heart of the VLSI CAD tools is a set of source programs obtained from Stanford University which combine to generate the necessary CIF file for design fabrication (using the nMOS process) from an input in the form of a Chip Layout Language (CLL). "CLL is a simple chip layout language, intended as an alternate to Caltech Intermediate Format for direct coding of layouts." [Ref. 5, p. 1] CLL is

a high level language which makes the task of designing a VLSI circuit easier than direct coding in CIF. It is written in the "C" language.

CLL has the following advantages:

- 1) It encourages bottom-up design by allowing small circuits to be developed and tested prior to being called by the "central" program.
- 2) It simplifies the design process by using defined commands which are easily memorized and used.
- 3) It allows calls to stored library cells which have been validated and tested.
- 4) It takes advantage of redundancy by allowing iteration of successive calls.
- 5) It is intended for lambda-grid design which eliminates the problem of changing scales.

The disadvantages of CLL are that it is capable of right-angle designs only and that it depends heavily on the designer's ability to develop a layout which is logically correct and geometrically compact.

The processor for CLL is the program **c1l**. As can be seen in the manual page for **c1l** (Appendix B), its basic function is to process the cell library externals and user written *.c1l* files to create a *.cif* file in CIF. The library externals are a set of designed cells in CIF which have been combined into one library file. The objective recommended for the designer is to construct various files in CLL format using this library (if needed), along with custom designed cells, and then use the CLL processor (**c1l**) to obtain check plots and a complete CIF file in output format. The specifics of this task and the use of the defined options for all of the source programs will be explained later in this thesis.

The CLL processor uses several programs to complete its tasks. Generally, these programs operate automatically through control of the CLL processor, which consists of a command program (**c1l**) and a CLL compiler (**c1l2**). Following is a list of these programs and their general functions. The manual pages

for these programs, which are given in Appendix B, can be called with the **man** command.

a. **Cif**

The **cif** command will cause the input *.cif* file to be converted to a cifout format so that it can be plotted at the GIGI terminal or the Versatec plotter. That is, it converts a file from CIF to binary form. This program can be run independent of **cil**. The cifout file format (*file.co*) is documented as CIFOUT in Appendix B.

b. **Cifload**

Cifload is called by **cil** (or can be used independently) to concatenate all of the *.cif* files that are given as input along with any library CIF files that are needed to produce the total CIF file.

c. **Merge**

For cases requiring the merge (or concatenation) of cifout files (sorted or unsorted), the **merge** program is used. The sorted cifout file is labeled *file.sco* while the unsorted cifout file is given the label of *file.co*. **Cil** uses **merge** to combine the sorted cifout file of a design with the unsorted cifout file generated by the Design Rule Checker (DRC). The program can also be used independently to merge several *.sco* files for a combined plot.

d. **Rplot**

Rplot is the program that allows the designer to plot the design (or part of it) on the Versatec plotter. When used by **cil**, the sorted cifout file is generally scaled to a size that can be plotted by **window**.

If used independently, **rplot** option **-i** will scale the plot. This program can be used to plot any sorted cifout file for geometric and design rule visual verification.

e. **Rsort**

In order to sort a cifout file so that it can be plotted, **cil** calls **rsort**. Any cifout file that is to be plotted must be sorted by x-coordinate due to the requirements of **rplot** and the Versatec.

f. **Tplot**

The **tplot** program generates a terminal plot (in color) on the GIGI terminal. The input must be labeled as a sorted cifout file (*file.sco*) even though the terminal does not require a sorted input. **Tplot** can be called independently or through **cil**. It should be used for quick checks of modifications to a design (since the Versatec plot is time consuming). **Window** is also called to scale the plot if **tplot** is used through **cil**.

g. **Window**

Both **rplot** and **tplot** receive scaled data from **window** when they are called by **cil**. (This occurs only if the **-i** option is used.) Additionally, **window** is capable of picking a selected portion of the total cifout file for plotting. This is particularly handy for a detailed plot of a small section of a large chip design.

2. Supporting Programs

In addition to **cil** and its associated programs, there are a few programs that can be used to aid in the completion of a VLSI design. These programs run independently of **cil**. They are documented in Appendix B and can be called with the **man** command.

a. **Cifar**

The **cifar** program allows the designer to develop an archive of CIF cells (files). This is very helpful in creating a library of "custom" cells which can be called in the main design file. This library can then be made a permanent record for future designers.

b. **Convert**

In order to make any sense out of a cifout file (sorted or unsorted), the file must be converted from its binary form into ASCII form using **convert**. This program can be used to find a problem in a cifout file that will not plot. The output should be directed (>) into another file name. To get this file back into cifout format, it must be converted back into binary form using **unconvert**.

c. **Plagen**

Plagen is a program that allows generation of a Programmed Logic Array (PLA) from a set of input and output specifications. Since a PLA is a very "regular" circuit, it can be used effectively in VLSI design. The PLA can be generated directly with **plagen** or indirectly using **plague**. The output is a CIF file which can be used as an external file to be called by the main design file.

d. **Plague**

To generate a PLA using output equations, use **plague**. This function converts the output equations of a PLA to the required inputs for **plagen**. Therefore, its output is usually **pipelined** directly into the **plagen** program.

e. **Unconvert**

Unconvert is used in conjunction with the **convert** program. It converts an ASCII file into binary cifout format. Its only use is a conversion after **convert** has been used to read a binary file.

3. Design Rule Checker (DRC)

As the feature sizes of integrated circuits diminish, greater importance must be placed on design rules. Separation and width errors could easily prove disastrous in an otherwise functional circuit. For this reason, the Design Rule Checker (DRC) should be used to indicate any design rule errors.

After a circuit has been designed and a sorted cifout file has been produced (using **cil**), the designer can determine the presence of most design rule

errors using the DRC. The DRC specifically checks for minimum separation and width errors within each layer. To do this, it must first determine any connectivity within the circuit. (Two poly rectangles side by side are a larger rectangle, not a minimum separation error.) However, it does have its limitations. For example, if two rectangles on the same layer cross, a short will exist which is not detected by the DRC. Additionally, if a contact is not fully connected, spurious errors may result.

The DRC processor is **drc**. It is documented in its manual page (Appendix B). Again, this tool is composed of several source programs which perform various tasks of this extensive design tool. Since the source programs run automatically and are not used independently, they will not be covered in this thesis.

As described in the manual page, the output of **drc** is stored in *file.drc*. Additionally, a cifout file is generated (*file.co*) which can be merged with the sorted cifout file and plotted on the Versatec or terminal so that the designer can locate the source of any design errors. The normal procedure is to determine the source of any design rule errors, correct them, and then run the **drc** program again. Once all errors have been eliminated, the circuit can be simulated.

4. Circuit Extractor

Prior to simulating a generated CIF file, it is necessary for the designer to convert the file into a representative circuit with defined node numbers (or names). This is required of most simulators. To do this, there are three programs that are used. Each of these programs is a *c-shell* file (Appendix A) which calls one or more source programs to complete its task. Since these source programs run automatically and are generally not used independently, they will not be covered in this thesis. The three *c-shell* programs are documented in Appendix B and are further explained in the following paragraphs.

a. **Extract**

The first step in circuit extraction is to convert the CIF file into a circuit with associated node numbers as reference points. These node numbers are assigned by **extract** with each different logic level having a separate node number. **Extract** also generates the necessary output files for plotting. A plot can then be made using **node-plot**.

b. **Node-plot**

The plotting function for an extracted circuit is **node-plot**. The output is a Versatec plot (the terminal cannot be used) of the designed circuit with nodes labeled. The dot (stipple) pattern for the different layers is not the same as that of the **cil** output, but this should not cause much confusion.

The purpose of obtaining this plot is to allow the designer to define node names to be used in circuit simulation. Once the numbers for the desired nodes have been obtained, they can be combined with appropriate names in the *.sym* file. As a minimum, the power node (*vdd*) and ground node (*gnd*) must be defined.

c. **Sim**

The last step in the circuit extraction is to generate the simulation file (*.sim*). **Sim** is a *c-shell* program (Appendix A) which converts the *.sym*, *.node*, and *.cap* files into the format required by simulators. The output *file.sim* file can then be checked using the Static Checker (**stat**) or simulated using the Event Level Simulator (**esim**).

5. Static Checker

Prior to trying an actual simulation, it is generally advantageous to put the design through a Static Checker. **Stat** will perform "two major tasks, checking gate ratios and looking for switch logic driving switch logic, as well as other tasks of less importance." [Ref. 6, p. 7] This step in the design process gives the

designer more confidence in the design of new cells but is of little use if only library cells were used. **Stat** is documented in Appendix B.

6. Event Level Simulator

After a designed circuit has been "extracted," it can be simulated using the processor for the Event Driven Switch Level Simulator (**esim**). This is an important step in the design process since it assures the designer that the circuit will perform to logic specifications. Although this simulator does not test for timing problems (such as rise and fall times, or race conditions), it does provide a good test of the logical accuracy of the designed circuit.

Esim is an interactive program which expects inputs from the user. The allowed inputs are described in detail in the manual page in Appendix B. After the design has "passed" its simulation, it is ready for manufacture.

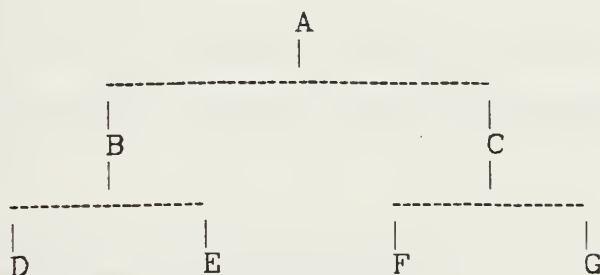
IV. GENERATING CIF USING CLL

Once an integrated circuit (chip) has been designed, simulated using a logic simulator (eg, ILOGS), and hand drawn as a geometric layout on lambda grid using the methods established by **Mead and Conway** [Ref. 1], the designer converts the chip into the Caltech Intermediate Format (CIF). This format is required by most fabrication facilities. For anyone who has attempted a direct conversion from layout geometry to CIF, the task is tedious, time consuming, and very susceptible to mistakes. An alternative route is to use the Stanford Chip Layout Language (CLL).

The CLL processor (**cll**) produces an output CIF file that can be sent directly to a fabrication facility. However, since it does not automatically check for design errors, additional checking is required. If the designer adheres to the following guidelines, the task of converting a geometric layout into a CIF file is much easier and less time consuming.

A. TUTORIAL

"CLL is a symbol-oriented language"[Ref. 5, p. 4] This means that the designer should divide his chip into a hierarchical structure and build each level as separate units with the "higher level" units combining the "lower level" units. The "top level" unit then creates the entire chip. For example, consider the following structure:



The units (symbols) D,E,F,G are converted to CIF first. D and E are then combined (possibly with additional geometry) by symbol B while F and G are combined by symbol C. The total chip is created with a "call" of B and C by the "main" file, A.

To allow this hierarchical "bottom-up" approach, CLL uses a high level language format which is an extension of the programming language "C." Each symbol, as well as the "main" chip file, will be a "C" file (Appendix A gives this format.) written by the designer using CLL and given a label of *file.cll*. The CLL processor can then be used to convert the symbols into CIF.

The following tutorial lists the allowed CLL statements (commands) and indicates their usage. (See Appendix C for a summary of these commands.)

1. Header

The top portion of a *.cll* file consists of statements which define the constants to be used in the file or link other symbols to the file. The allowed statements are:

a. Comments

Comments are used to make the file more readable. The format is

```
/*    [comments]    */
```

There is no restriction on the location of comments; however, comments cannot be nested.

b. External Symbols

CLL allows the definition of external symbols. This is useful in defining the symbols of the cell library or any *.cif* files generated by **plagen**. The format is

```
external name(cif bounds llx, lly xlen, ylen)
```


This statement defines any reference to *name* as being a CIF symbol with number, *cif#*. The smallest box which can be drawn around this symbol (bounding box) has its lower left corner at (llx, lly) on a lambda grid. The length in the x direction is *xlen* and the length in the y direction is *ylen*.

c. Defines

To define global constants which will be used in the body of the file, use the format

```
# define name value
```

where *#* must be in column one of the file. Any reference to *name* causes replacement by *value*. This *value* can also be an expression enclosed in parenthesis. For example,

```
# define lrx ( $llx + xlen$ )
```

gives the value of $llx + xlen$ every time *lrx* is stated.

d. Includes

```
# include file-name
```

includes the stated file along with the present file. This acts in the same manner as "linking" two files together or actually combining both files into one.

e. Conditionals

With complex designs, using many levels of hierarchy, there will be much linking of files together. This complexity can lead to coding errors due to statements involving undefined symbols. Additionally, because individual files may share the use of simple cells (e.g., drivers), there must be a way to prevent re-defining a cell within a linked file. The following formats can be used to

eliminate these problems:

```
# ifdef name
    statement
    .      (included only if name
    .      has been defined)
.
# endif

or

# ifndef name
    statement
    .      (included only if name
    .      has not been defined)
.
# endif
```

2. Symbol Definition

The symbol definition includes a symbol name (25 or fewer characters), an optional CIF number, and an optional bounding box. The format is

name [(*cif*# bounds *lx,ly xlen,ylene*)]

Normally, the CIF number and bounding box information are omitted since **cll** computes this information. If a CIF number is specified, subsequently encountered symbols are given CIF numbers sequentially from that number.

3. Body

The bulk of the *.cll* file is contained in the body. The opening brace ({) indicates that CLL statements are to follow and the closing brace (}) indicates that the file is ended. Positioning of these braces and of the statements is not critical to **cll**. However, indentation is recommended for ease of debugging. All statements must end with a semi-colon (;).

The following statements are allowed in the body:

a. Comments

Comments can be used in the body to make the file more readable.

They follow the same format as described in the paragraph under *Header*.

b. Rectangles

The CLL statement for defining a rectangle is

```
rect llx, lly xlen, ylen [layer];
```

This statement produces a box whose lower left corner is at (*llx, lly*) on a lambda grid and whose upper right corner is at (*llx+xlen, lly+ylen*). The layer of the rectangle is *layer* if it is specified. The default layer is the last layer defined in a **layer** statement.

c. Layers

A default layer can be defined with the statement

```
layer;
```

This statement is especially useful if the file is to have several rectangles on the same layer. This default layer is used any time that an optional layer is not specified and remains the default layer until changed with another **layer** statement. *Layer* can be any one of :

```
diffusion,diff,green  
poly,red  
metal,blue  
contact,cut,black  
implant,yellow  
glass  
metal2  
poly2
```


d. Wires

Wires are used to connect distant points within a symbol. The basic CLL statement is

wire [*layer*] *x,y wirelist*;

Again, *layer* is optional with omission implying use of the default layer. The starting point of the wire is indicated with *x,y*. *Wirelist* controls the path and size of the wire and can be one or more of:

<i>x,y</i>	move to (x,y)
<i>layer</i>	change layer
<i>u #</i>	up # lambdas
<i>d #</i>	down # lambdas
<i>r #</i>	right # lambdas
<i>l #</i>	left # lambdas
<i>x #</i>	move to (given x,same y)
<i>y #</i>	move to (same x,given y)
<i>w #</i>	set width to # lambdas

CLL requires a space between each entry of the list. A change in wire layer from metal to diffusion (or vice versa) or from metal to poly (or vice versa) causes automatic generation of a "via". Changes in wire layer from poly to diffusion cause generation of a "butting contact" which is not recommended. If the width is not specified, the default width for that layer will be used. The default width for any layer is the minimum allowed width for that layer.

An example of a wire statement is

wire 100,70 u 10 r 50 diff r 10;

This wire would start at (100,70) in the default layer, then it would move up 10 lambdas and right 50 lambdas before changing layers to diffusion and moving right 10 lambdas.

e. Vias

A via is a connection between metal and diffusion or metal and poly. This feature is useful in interconnecting input/output pads with their associated circuit points and for running wires that must cross paths within the chip. The format is:

via llx, lly *layer*;

The result is a 4 lambda square of metal, a 2 lambda square cut, and a 4 lambda square of *layer* whose lower left corner will be at (llx, lly) .

f. Calls

In order to invoke a defined symbol, a symbol call is used. It has the form

***name* $(llx, lly$ [*transformations*]);**

Since CLL always places the lower left corner of the symbol's bounding box at (0,0), placement of the symbol requires a shift in location (and possibly a rotation). The call first performs any *transformations* on *name* and then locates its lower left corner at (llx, lly) . All transformations leave the lower left corner of the bounding box at (0,0). Allowed transformations are:

flip ud	flip up-down
flip lr	flip left-right
flip rl	flip right-left
rotate 0	rotate 0 degrees
rotate 3	rotate 90 degrees clockwise
rotate 6	rotate 180 degrees clockwise
rotate 9	rotate 270 degrees clockwise
rotate 12	rotate 360 degrees clockwise

For example,

cell(100,500 flip lr rotate 9);

flips *cell* left to right, rotates it 270 degrees clockwise, and then places the lower left corner of the resulting bounding box at (100,500).

g. Iteration

When a symbol or cell must be repeated several times in a consistent fashion (as with adding drivers to a PLA), the symbol can be iterated using

iterate *nx,ny* [*xpitch,ypitch*] *symbol-name* (*llx,lly* [*transformations*]);

In this statement, *nx* indicates the number of times to replicate *symbol-name* in the x direction and *ny* indicates the number of duplicates in the y direction. *X-pitch* and *y-pitch* indicate the x and y spacing respectively. Either, or both, can be replaced with *default* to indicate that the bounding box dimension should be used for the spacing. If neither pitch specifications are stated, the default will be the bounding box dimension in both directions.

h. Expressions

As in the "C" programming language, numbers in CLL can be replaced by constant expressions. The allowed operators for these expressions are:

--	unary minus
-	subtraction
+	addition
*	multiplication
/	division
%	modulo

In addition, there are 4 special operations:

<code>dx(symbol-name)</code>	return width of a symbol
<code>dy(symbol-name)</code>	return height of a symbol
<code>pwidth(expr)</code>	return width of metal (where "expr" is in milli-Amps)
<code>!expr</code>	return cursor location plus "expr"

See Reference 5 for examples.

i. Print

For debugging a problem file, the statement

`print(expr)`

can be used. This will cause the value of *expr* to print out on the terminal.

B. CELL LIBRARY

Reference 7 provides a description of the basic library cells developed at Stanford University. These cells can be "called" using a *symbol call* if the following statement is placed in the header of the *.cll* file.

```
# include "/vlsi/lib/local/s_ext.cll"
```

The *s_ext.cll* file is a CLL file which "defines" the individual cells in the library. (The actual CIF file for the library is stored in "/vlsi/lib/libs.cif.")

C. USING CLL

There are several methods by which the designer can prepare a geometric layout prior to coding in CLL. This thesis will not discuss any of these; however, it is imperative that the design be logically correct and that it consist of only nMOS rectangular shapes assembled at right angles on a lambda grid.

If the design contains any PLA's, the CIF files for these can be generated directly using **plagen** or **plague**. These CIF files should be labeled in the form of

file.cif. Since the **plagen** function does not produce input or output drivers for the PLA, a CLL file (*file.cll*) has to be created for this addition. This CLL file must "call" the PLA CIF file and then attach the drivers.

In order to use the CLL processor (**cll**), a *.cll* file must be created for all of the symbols of the design and for the "main" program. Although it is possible to include more than one symbol in a single *.cll* file, this is not recommended. For simplicity and ease of construction, the designer should build separate files for each symbol. The following paragraphs describe the procedures for creating these files and producing the desired CIF files.

1. Making Files

The first step in using CLL is to create a *.cll* file for each symbol. This file can be created using the **vi** editor and must be labeled in the form of *file.cll*. The basic form of this file is given in Appendix C, along with a summary of the CLL commands.

If the design was originally drawn on a lambda grid, the CLL file can be coded directly using CLL commands and the desired coordinates from the layout. The best approach is to place the major components of the symbol and then add the interconnecting wires. When the file is complete, **cll** can be used for plotting (for visual error checking) and for creation of the CIF files.

2. Plotting

Throughout the coding process, it is desirable to produce check plots so that the designer can verify correct positioning of wires and transistors in a symbol. This can be accomplished on the GIGI terminal or on the Versatec plotter using **cll**. In either case, the best procedure is for the designer to generate a sorted cifout file (*file.sco*) before doing the plot. (It is possible to plot directly without storing the sorted cifout data; however, duplicate plots are not as easy to produce.) To generate the *.sco* file, use the command

cII[*options*] *file.cII file 1.cif file 2.cif ...*

The options are listed in the manual page of Appendix B. The particular options for this operation are **-I**, and **-D**. *File.cII* is the symbol file and the *.cif* files are any CIF files that were generated by the designer and "called" by the CLL file. If library cells were used, the **-Is** option must be used. The output is placed in *file.sco*.

The fastest way to get a visual plot of a *.sco* file is with a terminal plot. The command

cII -T [*options*] *file.sco*

generates a color plot of the sorted cifout file if a GIGI terminal is used. The options of interest in this case are one or more of: **-i**, **-n**, **-x**, **-y**, and **-D** (Appendix B).

The other way to get a plot produces a hard copy on the Versatec plotter. This approach uses much computer time and should be avoided during heavy usage times. The command is

cII -P [*options*] *file.sco*

where the options can be one or more of: **-b**, **-g**, **-i**, **-n**, **-x**, **-y**, **-s**, **-S**, and **-D** (Appendix B). The stipple patterns are defined in Figure 1 of Appendix B.

3. Creating CIF

In general, a CIF file must be created only for: (1)node extraction of symbols; (2)the total design; or (3)fabrication of the design. In the first two cases, the command

cII -C [*options*] *file.cII file 1.cif file 2.cif ...*

creates a file labeled *file.cif* which is the CIF file combining all "calls" and "includes" of the *file.cll* file. The two options that can be used are **-D** and **-I** (Appendix B). If library cells were used, the **-ls** option must be used.

To generate the final CIF file (which can be fabricated), use

```
cil -F [options] file.cll file1.cif file2.cif ...
```

The output is a CIF file labeled *final.cif*. The options are the same as listed above.

D. EXAMPLE

As a simple example of the use of **cil**, consider a gray-code-to-binary-code converter with reset, using a PLA. If the inputs to the PLA are defined as *reset*, *gray*, and *x0* and the outputs are defined as *binary* and *X0*, then it can easily be shown that the output equations are

$$\begin{aligned} \text{binary} &= x0' \& \text{reset}' \& \text{gray} + x0 \& \text{reset}' \& \text{gray}' \\ X0 &= x0' \& \text{reset}' \& \text{gray} + x0 \& \text{reset}' \& \text{gray}' \\ &(' \text{ for complement, } \& \text{ for AND, } + \text{ for OR}) \end{aligned}$$

Note that the combination of *x0* and *X0* is the feedback requirement for determining the "state" of the PLA and that the input gray code is serial with the most significant bit (MSB) entered first. Since *X0* and *binary* are identical, it would have been feasible to use only one of them as both output and feedback; however, since the PLA generator produces an "even" number of outputs, they are both used.

In order to generate a PLA CIF file using the above equations, an input file must be created for **plague** and the resulting output "pipelined" into **plagen**. Figure 4.1 gives the input file *pla*.

To generate the PLA CIF file, the command


```
plague < pla | plagen > pla.cif
```

is issued. The result is a CIF file, *pla.cif* and a schematic file, *pla.schem*.

```
CIF# 901;  
in: reset gray x0;  
out: X0 binary;  
binary = x0'&reset'&gray+x0&reset'&gray';  
X0 = x0'&reset'&gray+x0&reset'&gray';
```

Figure 4.1. PLAGUE INPUT FILE (PLA)

Additionally, the terminal responds with information about the PLA:

```
(3 input 2 output 2 term PLA);  
(external pla (cif 901 bounds --15,0 100,31);
```

The first of these lines can be added to the *pla.cif* file for documentation purposes. The second line can be used as an "external" statement in a *.cll* file. Note that the bounding box of this PLA has its lower left corner at (-15,0) and the upper right corner is at (85,31). Figure 4.2 is the *pla.cif* file with the added documentation.

Since the PLA is computer generated and was not drawn on a lambda grid, it is necessary to plot it so that the drivers and interconnecting wires can be added. However, the file must be modified with

```
C 901 T 0,0;  
E
```

at the end of the file. (CLL requires this modification.) A command of

```
cll -ls pla.cif
```

```
(ext 12);
(ext 13);
(ext 14);
(ext 15);
(ext 16);
(ext 17);
(ext 18);
(ext 21);
(ext 22);
(ext 23);
(ext 24);
(ext 25);
DS 901 250, 1;
(3 input 2 output 2 term PLA);
C 14 T -15 16;
C 12 T 0 16;
C 13 T 0 24;
C 23 T -4 7;
C 23 T -4 15;
C 12 T 16 16;
C 13 T 16 24;
C 24 T 23 7;
C 23 T 12 15;
C 12 T 32 16;
C 13 T 32 24;
C 23 T 28 7;
C 24 T 39 15;
C 15 T 48 16;
C 16 T 48 24;
C 12 R 0 -1 T 77 16;
C 22 T 64 9;
C 21 T 64 20;
C 22 T 72 9;
C 21 T 72 20;
C 14 R 0 -1 T 77 31;
C 13 R 0 -1 T 85 16;
L NM;
W 4 -13,16 -13,29 61,29;
DF;
```

Figure 4.2. PLA.CIF

generates the *pla.sco* file. To plot this sorted file, use

```
cil -P -g5.5 -i15 pla.sco
```

Figure 4.3 gives the result of this command.

From Figure 4.3 and the documentation of the input and output drivers in Reference 7, a CLL file can be created to form the converter symbol. Figure 4.4 is a possible solution for this CLL file (*converter.cll*). A sorted cifout file can now be created. However, first the *pla.cif* file must be modified by removal of the two lines that were added previously. Then a command of

```
cil -ls converter.cll pla.cif
```

will generate the file *converter.sco*. This file can be plotted on the GIGI terminal for a quick check using

```
cil -T converter.sco
```

or on the plotter with

```
cil -P -i15 -g5.5 converter.sco
```

(Note that the resolution of the terminal may not be sufficient for the designer to detect errors with the whole converter plotted. To get better resolution on a specific area he should window the area using the *-x* and *-y* options.) Figure 4.5 gives the result of the Versatec plot.

Once the symbol has been visually inspected and all errors have been corrected, a CIF file can be created with

```
cil -C -ls converter.cll pla.cif
```

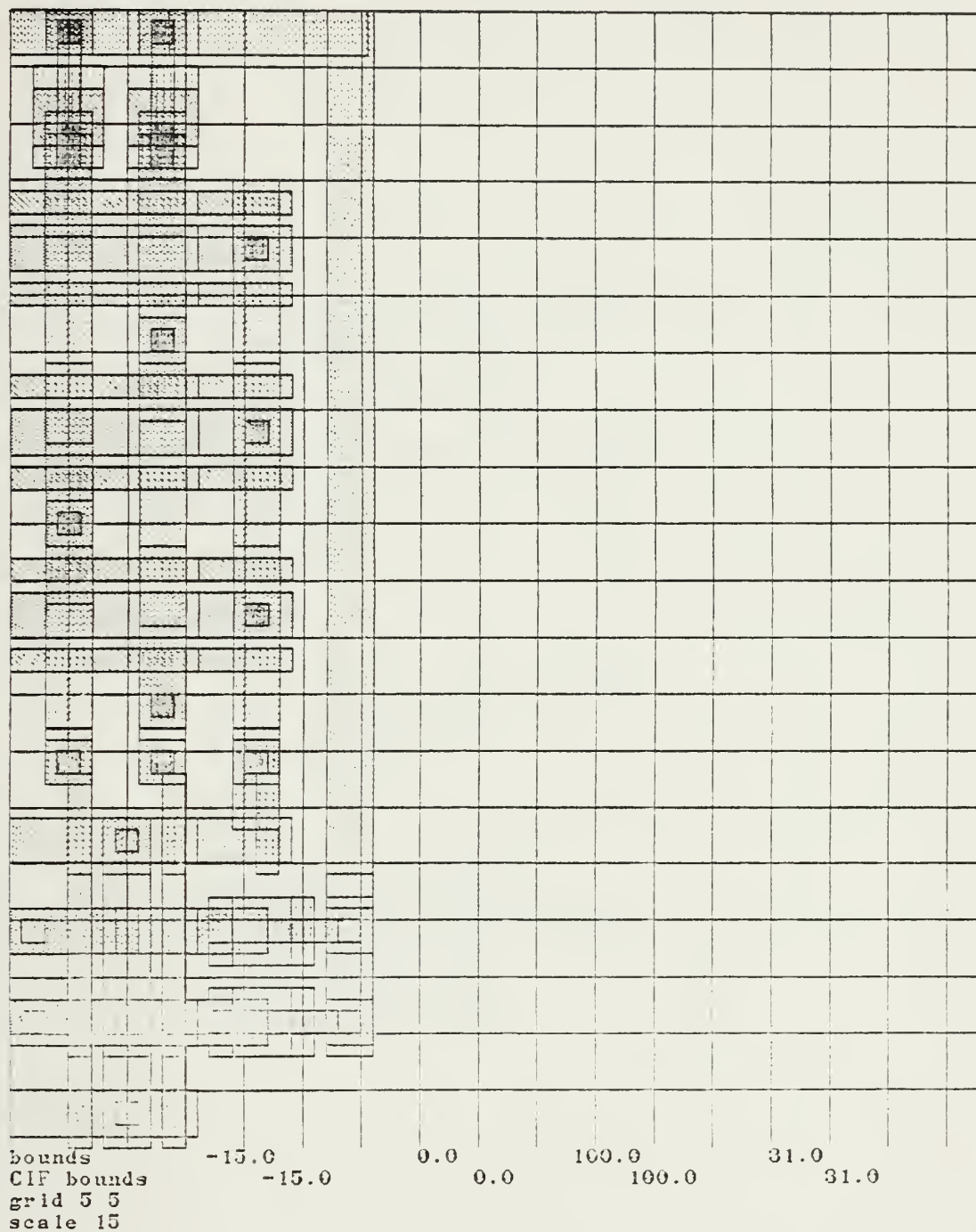



Figure 4.3. PLOT OF PLA.SCO

```
/* include external definitions for library cells */
#include "/vlsi/lib/local/s_ext.cll"

/* define external reference to pla */
external pla(cif 901 bounds --15,0 100,31)
gray_to_binary()
{
    /*place pla*/
    pla(0,0);

    /*attach input and output drivers*/

    iterate 3,1 PlaClockIn(15,--58);
    PlaClockOut(76,--53);

    /*connect gnd, vdd, and clock lines*/

    wire metal 2,1 w 4 d 23 r 14;
    wire metal 73,1 w 4 d 16 r 4;
    wire metal 62,--22 w 4 r 6 d 21 r 9;
    wire diff 57,--58 w 2 d 2 metal r 22 u 5 poly u 2;
    wire diff 25,--58 w 2 d 2;
    wire diff 41,--58 w 2 d 2;
    wire poly 87,--53 w 2 d 2;
}
```

Figure 4.4. CONVERTER.CLL

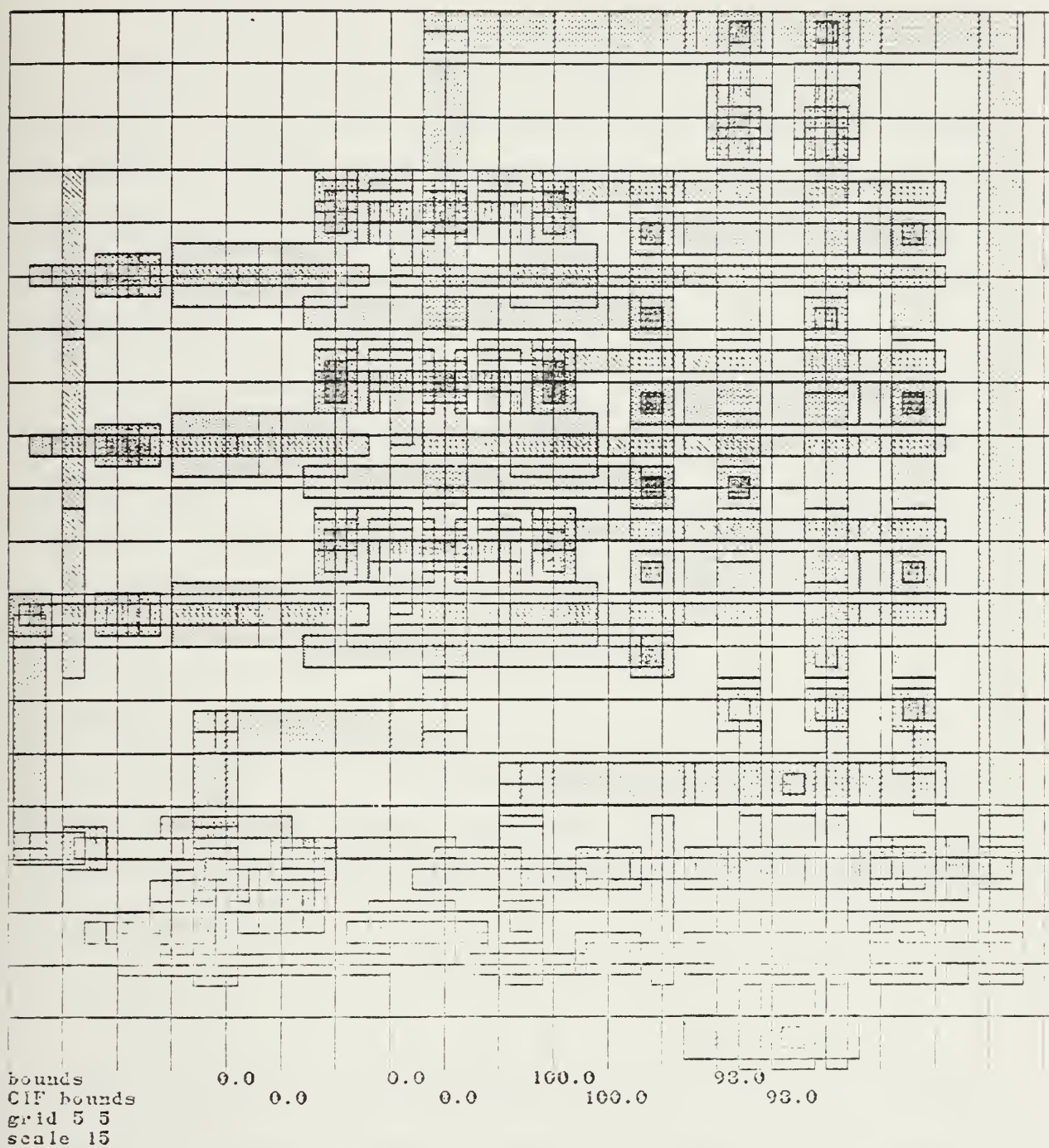


Figure 4.5. PLOT OF CONVERTER.SCO

V. DESIGN VALIDATION

Throughout the design process, the designer will wish to improve the confidence in his design using various CAD tools. These tools have been designed to overcome the most common weakness in creating a VLSI chip - human error. Although the perfect designer will have no need to validate a design against possible errors, most of us want to ensure that our finished product performs its desired function.

Use of the validation tools has no definite prescription. Each designer will have to decide as to what stage of the design process (what level of symbols) should be validated at a particular time. However, to keep the validation and debugging times to a minimum, we recommend that each "major" symbol be validated separately. The actual tools that can be used are the Design Rule Checker, the Static Checker, and the Event Simulator.

A. DESIGN RULE CHECKER

The Design Rule Checker inspects the sorted cifout file for design rule violations. In order to use the DRC processor (**drc**), the symbol must have a sorted cifout file (*file.sco*). The generation of this file is described in Chapter 4 of this thesis. A command of

drc *file.sco*

initiates the design rule check of the stated file. (See the manual page for DRC in Appendix B for details.) As with all of the validation tools, the DRC is time consuming and should be planned for times of low computer usage. The outputs of the DRC are *file.drc* and *file.co*.

1. Evaluation Of Outputs

File.drc provides a list of the lambda coordinates for any errors detected by the DRC processor. Each error is classified by type; however, one error may cause several coordinates to be listed. A comparison of the error coordinates with a plot of the symbol layout provides the designer with the source of the problem(s).

If the designer wishes to plot the output of the DRC combined with the actual layout, the command

```
cil -P [options] file.sco file.co
```

can be used. The result will be a plot of the symbol with "black" areas indicating the areas of design errors. Additionally, a terminal plot can be made using

```
cil -T [options] file.sco file.co
```

with "white" areas indicating the areas of design errors.

2. Example

As an example of the use of the DRC, consider the CLL file *wrong.cll* (Figure 5.1). This is the same as *converter.cll* (Figure 4.4) with the exception that the starting coordinate for PlaClockIn has been changed to (16,--58) vice (15,--58). This one lambda shift of the input drivers gives design errors for evaluation.

The first step in the validation process requires the generation of a sorted cifout file. The command

```
cil -ls wrong.cll pla.cif
```

will give the file *wrong.sco*. Now, to use the DRC, a command of

drc wrong.sco

is issued. (Prepare to wait, the DRC is extensive and slow.) Figure 5.2 gives the output file *wrong.drc* with a list of the design errors.

```
/* include external definitions for library cells */
# include "/vlsi/lib/local/s_ext.cll"
/* define external reference to pla */
external pla(cif 901 bounds --15,0 100,31)
gray_to_binary()
{
    /*place pla*/
    pla(0,0);

    /*attach input and output drivers*/

    iterate 3,1 PlaClockIn(16,--58);
    PlaClockOut(76,--53);

    /*connect gnd, vdd, and clock lines*/

    wire metal 2,1 w 4 d 23 r 14;
    wire metal 73,1 w 4 d 16 r 4;
    wire metal 62,--22 w 4 r 6 d 21 r 9;
    wire diff 57,--58 w 2 d 2 metal r 22 u 5 poly u 2;
    wire diff 25,--58 w 2 d 2;
    wire diff 41,--58 w 2 d 2;
    wire poly 87,--53 w 2 d 2;
}
```

Figure 5.1. WRONG.CLL

A merge of the sorted cifout file with the file *wrong.co* can be produced with the command

cll wrong.sco wrong.co

The output is stored in *merge.sco* and can be plotted on the Versatec or the

Poly min width errors:

16, 60
16, 60
17, 63
18, 62
24, 60
24, 60
25, 63
26, 62
32, 60
32, 60
33, 63
34, 62
40, 60
40, 60
41, 63
42, 62
48, 60
48, 60
49, 63
50, 62
56, 60
56, 60
57, 63
58, 62

Diff min width errors:

24, 4
25, 2
25, 3
24, 5
26, 2
26, 2
26, 3
26, 5
40, 4
41, 2
41, 3
40, 5
42, 2
42, 2
42, 3
42, 5

Metal min width errors:

Contact metal cover missing:

Contact poly or diff cover missing:

Poly to diff-contact separation error:

There are 35 transistors

Poly separation errors:

Diff separation errors:

Metal separation errors:

(continued on next page)


```

Poly to Diff separation errors:
18, 62
19, 61
23, 62
24, 61
34, 62
35, 61
39, 62
40, 61
50, 62
51, 61
55, 62
56, 61
Implant surround error:
Poly-Diff-transistor surround errors:
26, 3
42, 3

```

Figure 5.2. WRONG.DRC

terminal. Or, a plot can be obtained directly with the command

```
cil -P -g5.5 -i15 wrong.sco wrong.co
```

Figure 5.3 is the result of issuing this command and can be used to evaluate the errors detected by **drc**. The error points are indicated by "dark" blocks.

From Figures 5.2 and 5.3, the following errors can be seen:

Poly min width and Poly to Diff separation errors
in the areas of:

```

17,60
25,62
33,62
41,62
49,62
57,62

```

Diff min width & Poly-Diff transistor surround
errors in the areas of:

```

25,3
41,3

```

A closer look at the plot indicates that all errors are caused by the shift in placement of the input drivers and can be easily corrected by changing the "calling" coordinate of the input driver.

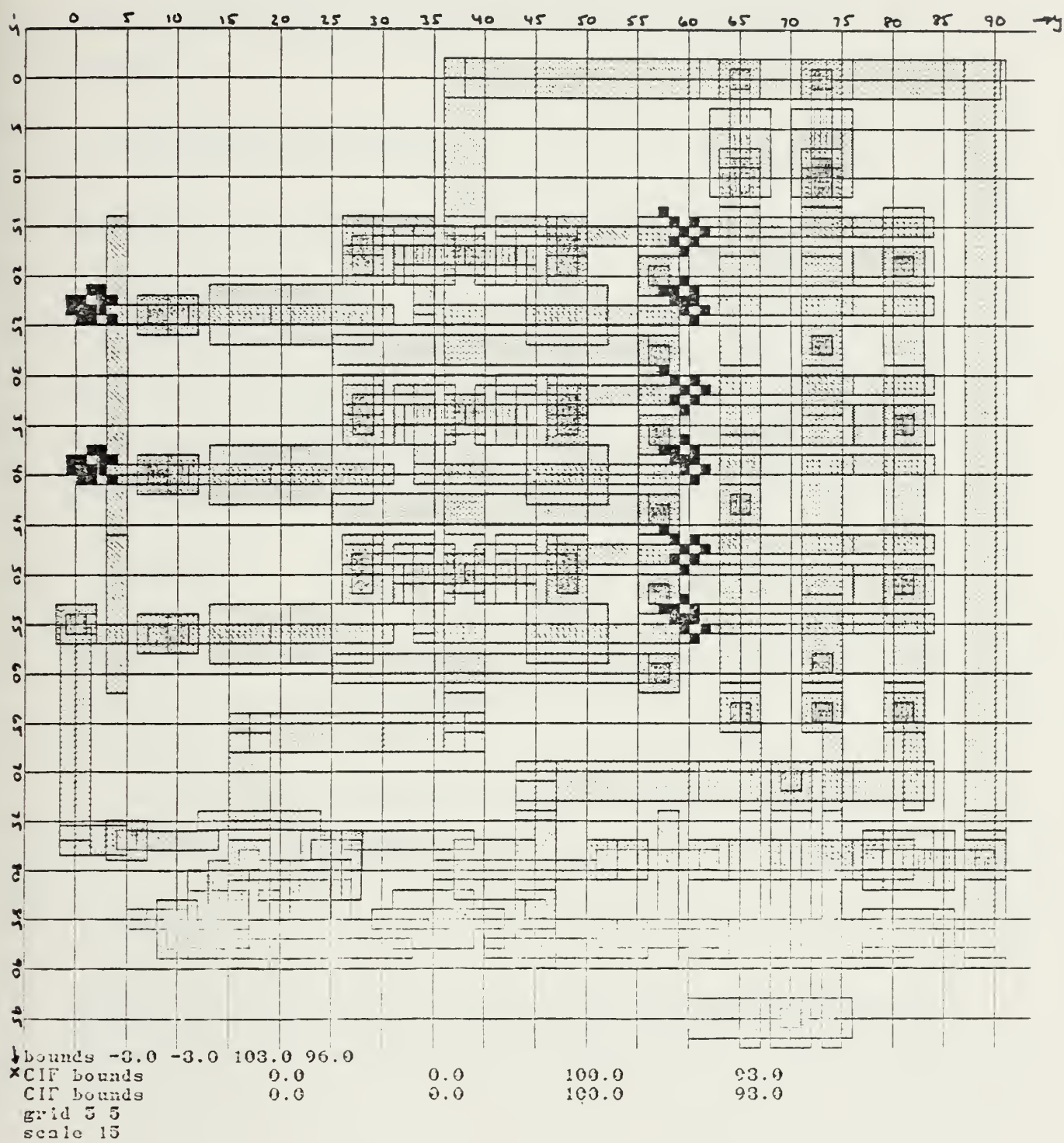


Figure 5.3 PLOT OF WRONG.SCO AND WRONG.CO

As an alternate method for determining the source of DRC errors, a terminal plot of the "window" around an error coordinate can be used. For example,

```
cil -T -x0.25 -y50.70 merge.sco
```

indicates one of the problem areas.

B. CIRCUIT EXTRACTOR

The Circuit Extractor extracts a circuit from a CIF file so that it can be simulated. The manual page for the Circuit Extractor (EXTRACT in Appendix B) gives the commands necessary to perform an extraction. If it is desired to extract a symbol for which there is a CIF file labeled *file.cif*, then the proper command is

```
extract file
```

The result of issuing this command will be generation of *file.def*, *file.cap*, *file.gate*, *file.node*, *file.rec*, and *file.sym*.

1. Plotting

A plot of the node numbers which have been assigned to the symbol can be obtained with the command

```
node-plot file
```

This produces a Versatec plot of the layout with stipple patterns as defined by Figure 2 of Appendix B and node numbers at various points throughout the plot.

2. Defining Nodes

In order to evaluate a circuit under simulation, it is necessary to assign names (labels) to selected nodes in the circuit. As a minimum, the power bus (*vdd*) and the ground bus (*gnd*) must be identified. To do this, the *file.sym* file has to be modified. (It is empty.) Referring to the **node-plot** output, the designer can use **vi** to modify the *file.sym* file by making a list of node numbers with their associated names. For example, the most commonly labeled nodes are power, ground, inputs, outputs, and clocks. Once a node has been given a name, it can no longer be referred to by number.

3. Creating A Simulation File

A simulation file (*.sim*) is required by both the Static Checker and the Event Simulator. If an extraction has been performed on a symbol *file.cif* and the *file.sym* file has been modified, then the function **sim** can be used to generate the simulation file. It is invoked with

```
sim file
```

The output is *file.sim* and can be used for static checking and simulation.

4. Example

The example of Chapter 4 can be extracted by issuing the command

```
extract converter
```

The extracted circuit can now be plotted with the command

```
node-plot converter
```

The terminal responds with


```
type-stipout /vlsi/tmp/converter.stip[A-A]--when ready
```

This statement indicates that the sorted data for the plot fits into a 240 lambda strip (A) and can be plotted with

```
stipout /vlsi/tmp/converter.stipA
```

Figure 5.4 is the result of issuing this command.

Using the node numbers as shown in Figure 5.4, a *.sym* file can be created to label the power, ground, input, output, and clock nodes. Figure 5.5 gives this *converter.sym* file.

The simulation file can now be created with

```
sim converter
```

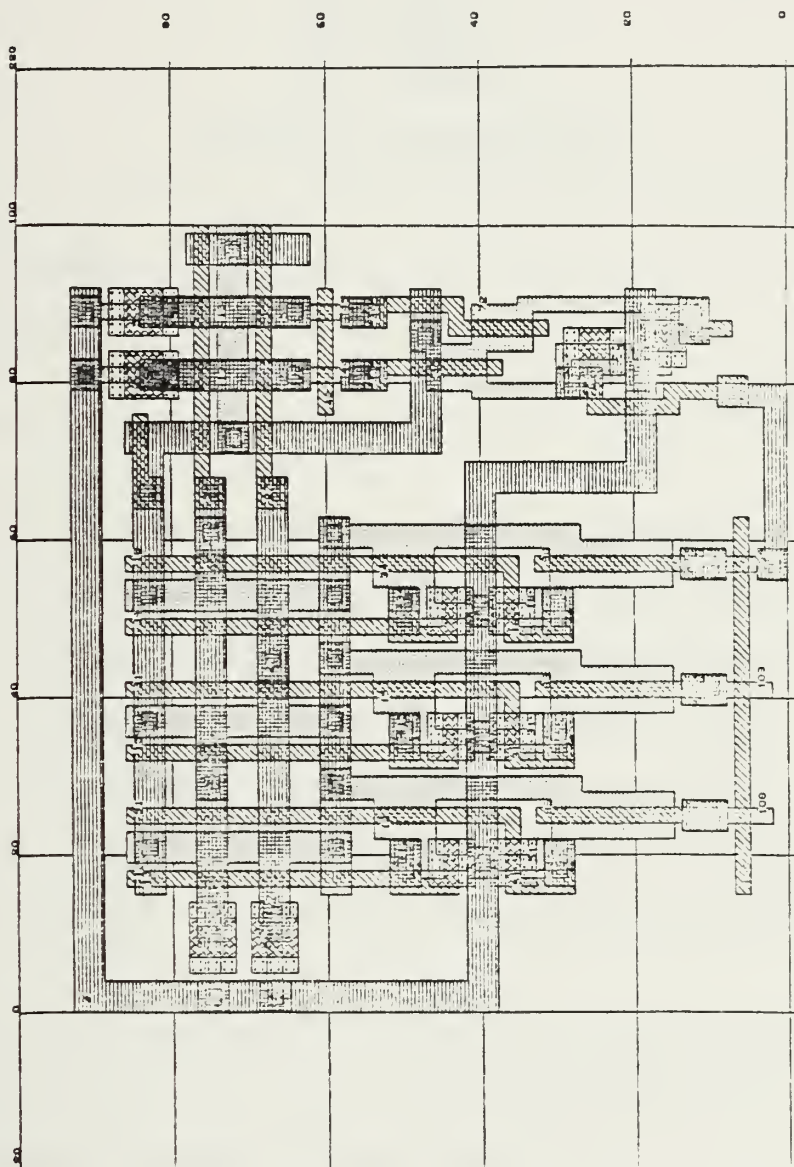



Figure 5.4. PLOT OF CONVERTER STIPA

13	gnd
3	vdd
105	phi1
42	phi2
108	reset
109	gray
72	binary

Figure 5.5. CONVERTER.SYM

C. STATIC CHECKER

The command

stat *file.sim*

initiates a static (dc) analysis of the symbol *file*. **Stat** accomplishes this by using a data base consisting of a set of transistors, each with a gate, source, drain, length, width, and type (currently enhancement, depletion, and intrinsic), and a set of nodes. The program tries to determine the threshold drops on the nodes and the use of the transistors by adding to this data base. Finally, having understood as much of the circuit as possible, the program makes a pass over the transistors that are still not understood and indicates any that are obviously incorrect.

As stated in the manual page (STAT in Appendix B), the two outputs of **stat** are *standard output* containing an entry for each potential error and *standard error output* containing the number of nodes, inverters, etc.

1. Evaluation Of Outputs

The manual page for **stat** (Appendix B) gives a list and explanation of the outputs. However, a few comments are necessary to prevent an incorrect analysis. Prior to beginning a static check, it is necessary to identify the power (*vdd*) and ground (*gnd*) nodes of the circuit. The Static Checker identifies input

nodes by locating any enhancement transistors whose gate is ground, source is the node, drain is ground, length is 2 lambda, and width is greater than 39 lambda. If no input nodes are found, the "Propagate" output will not be valid. Some of the errors are repeated in both the outputs. For example, a chip containing 5 ratio errors (all with a ratio of 3.45) would have one line in the *standard error output*, indicating that the ratio 3.45 occurred 5 times, and 5 entries in the *standard output*, one for each ratio error, detailing the specific nodes and transistors involved.

2. Example

Continuing with the *converter* example of Chapter 4, a command of

```
stat cconverter.sim > converter.stat
```

initiates the static analysis and stores the standard output in *converter.stat*.

Figure 5.6 is this file.

Evaluation of Figure 5.6 results in the following:

- 1) The unknown threshold drops are the input, clock, and output nodes. This is expected since there are no input or output pads.
- 2) The ratio messages indicating a pu/pd ratio of 0.00 are associated with the input and output nodes and are also expected due to the fact that no pads are present.
- 3) Since no pads are present, the Static Checker did not identify any input or output nodes, so that all other nodes are not affected by the input nor do they affect the output. (Indicated by the Propagate print out.)
- 4) The only nodes not affected by either ground or power are the input, clock and output nodes. This is desirable.
- 5) There is no indication of transistor (or other) errors, so we assume that the circuit is statically correct.

```

2136 bytes of 'free' storage used
Unknown threshold drop on node: 44 (80,58)
Unknown threshold drop on node: 45 (88,58)
Unknown threshold drop on node: 86 (24,32)
Unknown threshold drop on node: 88 (40,32)
Unknown threshold drop on node: 90 (56,32)
Unknown threshold drop on node: phi1 (15,6)
Unknown threshold drop on node: phi2 (76,60)
0.00 18 (50,36) <2x2>: <90?2x16> gnd
0.00 31 (34,36) <2x2>: <88?2x16> gnd
0.00 51 (18,36) <2x2>: <86?2x16> gnd
0.00 64 (83,26) <4x2>: <44?2x8> gnd
0.00 binary (84,16) <4x2>: <45?2x8> gnd
Propagate ( 10): 2 (78,85)
Propagate ( 10): 5 (86,85)
Propagate ( 10): 9 (7,77)
Propagate ( 10): 12 (16,85)
Propagate ( 10): 15 (32,85)
Propagate ( 10): 18 (56,85)
Propagate ( 10): 27 (7,69)
Propagate (0110): gray (40,4)
Propagate ( 10): 31 (40,85)
Propagate ( 10): 34 (48,85)
Propagate ( 10): 44 (80,58)
Propagate ( 10): 45 (88,58)
Propagate ( 10): 51 (24,85)
Propagate ( 10): 64 (79,46)
Propagate (0110): 86 (24,32)
Propagate (0110): 88 (40,32)
Propagate ( 10): 90 (56,32)
Propagate ( 10): binary (88,40)
Propagate (0110): reset (24,4)
Propagate (0110): phi1 (15,6)
Propagate (0110): phi2 (76,60)

```

Figure 5.6. CONVERTER.STAT

D. EVENT SIMULATOR

The Event Driven Switch Level Simulator (**esim**) can be invoked with

esim *file.sim*

Once invoked, **esim** performs as an interactive simulator with a prompt of

sim>

The manual page (ESIM in Appendix B) gives the allowed commands for completing a simulation.

1. Using Esim

Although it is possible to complete an entire simulation in the interactive mode, the designer will find that the most effective procedure is to plan a desired test sequence (a set of inputs and clock cycles to generate a known set of outputs) and use **vi** to create a "macro" file to be read by **esim**. This macro file should contain the initiating commands (as described in the manual page) for the input nodes and the desired clock sequences. Additionally, it should identify the nodes to be "watched." The file can be initiated either with

@ *file.macro*

after **esim** has been invoked or can be included at the start of the simulation with

esim *file.sim file.macro*

Two-phase clocks can be defined with the **K** command. For example,

K phi1 110000 phi2 000110

defines one cycle of a two-phase, non-overlapping clock. The input nodes can be set to a "high" or "low" state using the **h** and **l** commands respectively or with the **V** command. The outputs can be "watched" with the **w** or **W** commands.

2. Example

To simulate the *converter* example, the macro file *esim.macro* must first be generated. (See Figure 5.7.) The command

esim -esim.out converter.sim esim.macro

invokes the desired simulation and stores the output in *esim.out* (Figure 5.8).

```
w gray binary
K phi1 110000 phi2 000110
h reset
c
V reset 00001 gray 1110
R
V reset 00001 gray 1001
R
V reset 000001 gray 11100
R
V reset 000001 gray 01011
R
V reset 000001 gray 11111
R
q
```

Figure 5.7. ESIM.MACRO

```
23 transistors, 23 nodes (12 pulled up)
binary=0 gray=X
cycle took 32 events
>11101:gray
>01011:binary
>10011:gray
>01110:binary
>111001:gray
>010111:binary
>010110:gray
>001101:binary
>111111:gray
>010101:binary
23 transistors, 23 nodes (12 pulled up)
```

Figure 5.8. ESIM.OUT

Evaluating Figure 5.8, we see that the last (right) bit given for *gray* and the first (left) bit given for *binary* are invalid and are only typed because of the nature of the **R** command. There are five input gray codes used for the

simulation with the respective equivalent binary code outputs. The MSB is to the left of the printout. Comparison of each output with its input reveals that the converter works as designed. For example, an input gray code of 1110 should (and does) produce a binary output of 1011.

The last step in the design process is the combining of all valid symbols to form the total chip. The CIF file for the chip is formed with the **cil -F** command to generate *final.cif* as described in Chapter 4. This file should be validated with **drc**, **stat**, and **esim**. Once the total chip has completed all of the validation tests to the satisfaction of the designer, it can be manufactured with a good chance of success. Appendix D gives the procedures for DARPA supported chip manufacture.

VI. PROJECT: 16 BIT VERY FAST PIPELINED CARRY LOOK AHEAD ADDER

A. INTRODUCTION

This chapter discusses the development of an LSI design from conception through simulation. The creation of a sixteen-bit adder is a two part research project. First, the CAD tools and hardware resources available at the Naval Post-graduate School needed to be exercised and documented. Secondly, recent interest in the design of recursive (IIR) digital filters indicates a need for fast adders that possess a predictable amount of delay. The adder designed and fabricated in this thesis project has both of these attributes. Moderate complexity, testability, and the possible utilization in a larger complex system entered into the decision to create the 16 BIT ADDER.

B. LOGIC DESIGN

1. Pipelining

Nearly any digital circuit which accepts incoming data, processes the information and produces an output may be suited to pipelining techniques. As the title suggests, "pipelining" is incorporated into the design. Pipelining is a technique whereby a larger more complex functional unit is serially decomposed into several smaller less complex functional units. Each sub-functional unit accepts a digital input, performs a "sub-process" and outputs the "sub- result" to the following stage. The output of the final unit is the overall result. The term "pipeline" suggests data entering a pipe at one end and the result spilling out at the other end with processing taking place between the input and output. This type of visualization omits a very important parameter - timing. A better way to comprehend the pipeline operation is to visualize an assemblyline. The production of a automobile is a good example. Inputs are all the necessary components

and the output is the finished automobile. Along the assemblyline, certain functions are performed at each work station and a finite amount of time is required for each function to be completed. The system clock provides the control signals to the assemblyline. At each tick of the clock, the yet unfinished auto moves one additional station towards final completion. The fastest speed at which the assemblyline may progress is determined by the work station that requires the longest period of time to complete its task. If the line advances before a station completes its function, the output product is rendered unuseable.

The basic advantage of pipelining is increased speed of operation. Two disadvantages are complexity (additional hardware) and "fill" time. Continuing with the above example, fill time (delay) is the amount of time it takes for the first car to reach completion. The fill time is as large as, or larger than, the amount of time that is required by one group of workers to complete the entire automobile. Increased speed of operation is realized when the assemblyline is broken down into numerous easy tasks. This is the key point in pipelining. While it may take many ticks of the clock to obtain the first car, an additional auto is completed for every subsequent tick of the clock. And, the period of the clock is the time required to complete the short easy tasks. The complexity of such a line is obvious when compared to the "one group-all tasks" approach.

When referring to a digital circuit, the increased complexity arises from the necessity to store intermediate results between sub-units until the next clock signal is received. Normally, the additional number of storage devices is great and can quickly become as complex as the functional logic. The fill time is usually considered to be a disadvantage. However, this project originated as a sub-functional unit of an infinite impulse response digital filter. A precise amount of delay is required for this type of circuit. In this case, the fill time is considered to be an advantage. The highest clock rate can be realized when

there is only one gate delay per sub-unit. This maximum clock frequency depends upon the amount of time needed for the data to progress from the input storage device through a single level of functional gating and then stabilize the output storage device.

Carry-look-ahead addition, which is discussed in the next section, readily lends itself to pipelining techniques. The adder is divided into "N" sub-functional stages. Two sixteen-bit vectors are applied to the inputs. "N" clock cycles later the sum of the two vectors is obtained. If two new vectors are applied to the input of the pipe at each clock pulse, then after the initial fill time of "N" clock cycles, a resultant sum is output for each subsequent cycle. Many pipelined circuits use a two-phase non-overlapping clock. This allows for data to be clocked into a sub-functional stage on one phase and the result to be clocked out of that stage on the other phase. The limiting high clock rate is determined by the sub-unit with the largest delay. The limiting low clock rate is determined by the length of time the charge stored on any input/output dynamic register remains above threshold. Recall from chapter two that this charge decreases exponentially and is on the order of milliseconds. It is possible to utilize flip-flops as the input/output register to alleviate concern for the limiting low clock rate, however, this adds to the overall complexity of the design.

2. Carry-Look-Ahead Addition

The standard one-bit full adder utilizes three inputs and produces two outputs. It accepts the two binary bits to be added, denoted as A_i and B_i , and a carry input C_{i-1} from the previous stage. The outputs are the sum bit S_i and the carry output C_i . One set of boolean expressions that realizes a one-bit adder is shown in equations (6.1 and 6.2). An n-bit full adder is constructed by connecting (n) one-bit stages in parallel. Clearly, the amount of time for an n-bit full adder to produce the output depends on (n). For the worst case analysis,

$$S_i = (A_i) \text{ XOR } (B_i) \text{ XOR } (C_{i-1}) \quad (6.1)$$

XOR implies the eXclusive-OR logical operation.

$$C_i = \bar{A}_i B_i C_{i-1} + A_i B_i \bar{C}_{i-1} + A_i \bar{B}_i C_{i-1} + A_i \bar{B}_i C_{i-1} \quad (6.2)$$

input vectors \vec{A} and \vec{B} could be such that if a carry were generated by the least significant bits (A_0 and B_0), the carry would "ripple" through the remaining stages. The output would not be valid until this carry propagated through and affected all of the stages. As the size of the adder gets larger, this "ripple" effect induces a larger time delay. In order to increase the operating speed of n-bit adders whose length exceeds four bits, [REF.9:pg88] a technique known as **carry-look-ahead addition** is incorporated. Additional circuitry is added to produce two functions called "carry generate" and "carry propagate". The carry generate function (G_i) is true when a carry is generated in the i th stage regardless of the value of the carry into the i th stage. The carry propagate function (P_i) is true when the i th stage would propagate (pass) an incoming carry to the next most significant stage. The logic equations for the two functions are defined as:

$$G_i = A_i B_i \quad (6.3)$$

$$P_i = (A_i) \text{ XOR } (B_i) = A_i \bar{B}_i + \bar{A}_i B_i \quad (6.4)$$

When equation (6.4) is substituted into equation (6.1) the following is obtained:

$$S_i = ((A_i) \text{ XOR } (B_i)) \text{ XOR } (C_{i-1}) = (P_i) \text{ XOR } (C_{i-1}) \quad (6.5)$$

Equation (6.2) can be simplified to:

$$C_i = A_i B_i + (C_{i-1}) ((A_i) XOR (B_i)) \quad (6.6a)$$

Substituting equation(6.3) and equation(6.4)into equation (6.6a), C_i becomes:

$$C_i = G_i + P_i C_{i-1} \quad (6.6b)$$

Note that all P_i and G_i can be simultaneously determined from the input vectors \vec{A} and \vec{B} . And, equation(6.5)implies that S_i can be produced in parallel provided all the carry inputs can be obtained simultaneously. This can be accomplished by solving equation(6.6b). Equation (6.6b) is a recursive equation and with the carry input to the first stage being defined as C_{-1} , a set of carry equations for all C_i can be developed. The general solution is:

$$C_i = G_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + \dots G_0P_1P_2 \dots P_i + C_{-1}P_0P_1P_2 \dots P_i \quad (6.7)$$

To illustrate this formula, let $i=4$. Then, C_4 the carry bit needed to realize S_5 has the following logic equation:

$$C_4 = G_4 + G_3P_4 + G_2P_3P_4 + G_1P_2P_3P_4 + G_0P_1P_2P_3P_4 + C_{-1}P_0P_1P_2P_3P_4 \quad (6.8)$$

S_5 is obtained by combining P_5 and C_4 in an XOR gate. Equation (6.7) is one method of realizing a carry-look-ahead (CLA) "unit" An n-bit adder requires equation(6.7) to be expanded to $(i=n-1)$. Two other units are necessary to produce a carry-look-ahead "adder". The carry generate/propagate unit produces the G_i 's and P_i 's from the input vectors \vec{A} and \vec{B} . Both the G_i 's and P_i 's are then delivered to the input of the (CLA) unit which forms the carry bits. These carry bits are combined with the P_i 's in a third unit called the

summation unit to produce the S_i 's. This type of carry-look-ahead adder (even though it has three main units) is called a one-level CLA adder. The distinguishing difference between a one-level CLA adder and a two-level CLA adder is the number of CLA units located between the carry generate/propagate unit and summation unit.

When (n)-the number of bits to be added-grows large, the problem of fanout becomes critical. Note that in equation(6.8) for $i=4$, P_4 appears in five product terms. For a sixteen-bit adder, the most significant bit is $i=15$. To produce S_{15} , P_{15} and C_{14} are required at the input of an XOR gate. It is seen from equation(6.7) that P_{14} would occur in fifteen product terms to develop C_{14} . This implies that the P_{14} output of the carry gen/prop unit experiences a fanout of fifteen upon entering the CLA unit. A fanout of fifteen causes current limitations to be approached or exceeded in some large scale integration technologies [REF.9:pg.88]. To circumvent this problem, a two-level carry-look-ahead adder is utilized. An additional CLA unit is inserted between the gen/prop unit and the summation unit. The addition of this extra CLA logic alleviates the fanout problem but introduces extra gate delays and increases the overall complexity of the adder. Modifications are made to this additional CLA unit to develop what is called a "block" carry-look-ahead unit or (BCLA). The BCLA unit is very similar to the CLA described by equation(6.7). "Block" denotes a definite bit length. Normally, (n) is equal to four or eight. Instead of producing (for $n=4$) four carry bits, the BCLA unit develops the three least significant carry bits along with two other signals denoted as "block carry propagate" and "block carry generate". An example delineates the differences between the two types. A standard four-bit CLA unit produces $C_{i+3}, C_{i+2}, C_{i+1}, C_i$; $i=0,4,8,12,\dots$ while a four-bit BCLA unit develops $BP_j, BG_j, C_{i+2}, C_{i+1}, C_i$; $j=0,1,2,\dots, i=4j$. BP and BG signify block carry propagate and block carry generate. The BP_j bit is

valid if a carry into the block results in a carry out of the block. For $n=4$, BP_j is determined by the following:

$$BP_j = P_{i+3}P_{i+2}P_{i+1}P_i ; j=0,1,2,3,\dots ; i=4j \quad (6.9)$$

The BG_j variable is true if the carry out of a block was produced within that block. BG_j can be written as:

$$BG_j = G_{i+3} + G_{i+2}P_{i+3} + G_{i+1}P_{i+2}P_{i+3} + G_iP_{i+1}P_{i+2}P_{i+3} \quad (6.10)$$

$$j=0,1,2,3,\dots ; i=4j$$

Using equation(6.6b) and the recursive solution method, the block carry bits can be formed from the following equations.

$$BC_3 = BG_0 + C_{-1}BP_0 \quad (6.11)$$

$$BC_7 = BG_1 + BG_0BP_1 + C_{-1}BP_0BP_1$$

$$BC_{11} = BG_2 + BG_1BP_2 + BG_0BP_1BP_2 + C_{-1}BP_0BP_1BP_2$$

$$BC_{15} = BG_3 + BG_2BP_3 + BG_1BP_2BP_3 + BG_0BP_1BP_2BP_3 + C_{-1}BP_0BP_1BP_2BP_3$$

Note that equation(6.11) is a recursive formula on a block level where equation(6.6b) is a recursive formula on a stage level. For a sixteen-bit adder, BC_{15} is the carry out or overflow bit.

A sixteen-bit two-level carry-look-ahead adder constructed from commercially available four-bit CLA/BCLA units is shown in Figure (6.1). Inspection of Figure 6.1 and the equations presented in this section yield the following gate delay analysis. The propagate/generate unit requires three levels of gating due

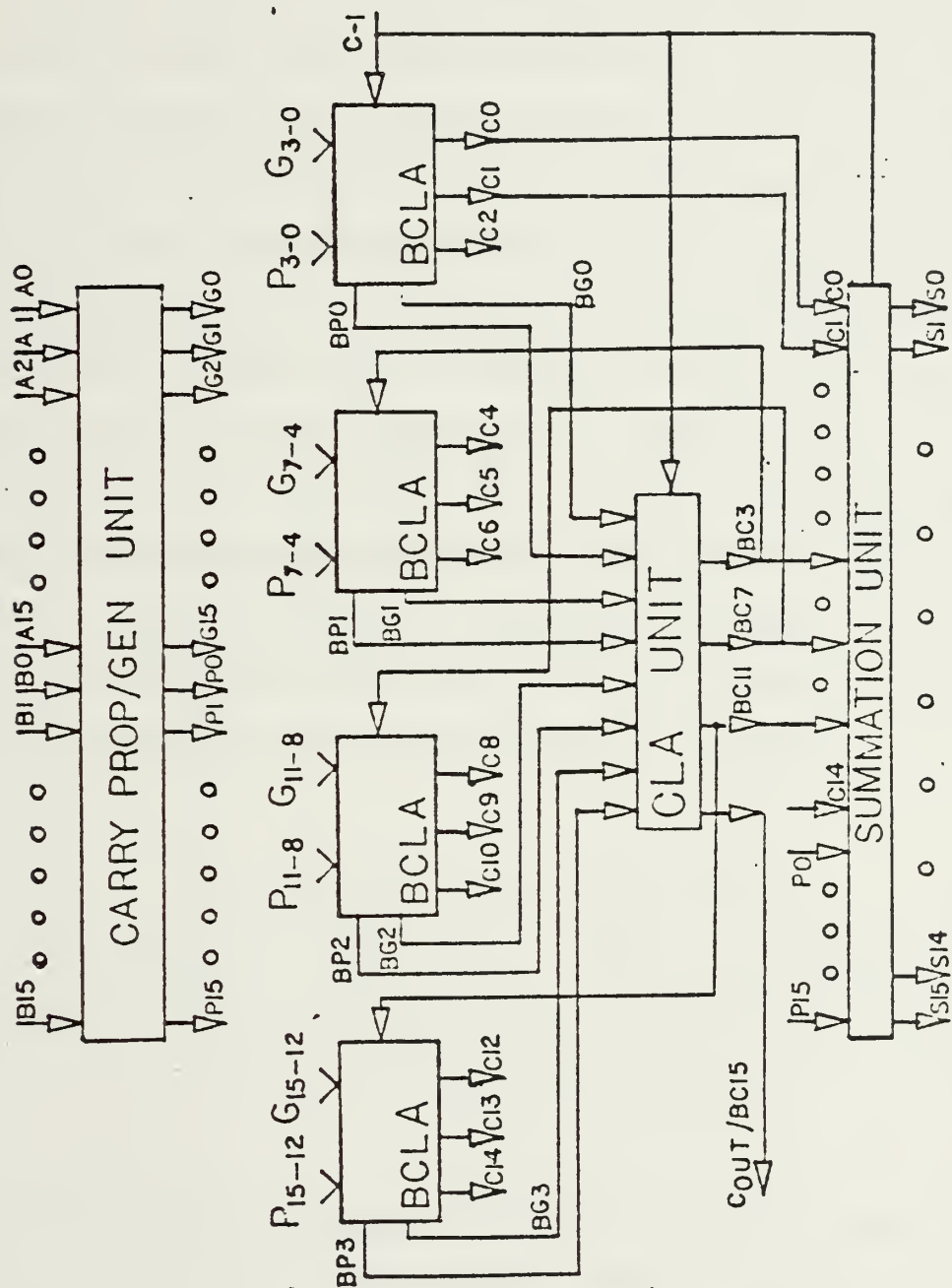


Figure 6.1 Two level 16 bit CLA adder

to the XOR operation needed to realize the P_i 's and the G_i 's. Three levels of gating are also needed by the summation unit for the same reason. It produces S_i from the logical XOR of P_i and C_{i-1} . The block carry-look-ahead unit performs two functions, each of which requires two gate delays. First, the BP_j 's and the BG_j 's are developed in accordance with equations (6.9) and (6.10). "After" the BP_j 's and the BG_j 's are formed the BC 's are realized by eq(6.11). This formation of the BC 's takes place in the standard CLA unit and therefore introduces two gate delays. The BC 's are then passed on to the next most significant BCLA where another two gate delays are required to develop the remaining carry terms. The two-level adder requires a total of twelve gate delays whereas the one-level adder (due to the absence of the BCLA level) requires only eight gate delays. The equation for the remaining carry terms is developed by using equation(6.7) with slight modifications and limiting the subscripts to certain allowed values. Each BCLA produces only three carry bits. Redefining C_{-1} as BC_{-1} , the carry bits for a sixteen-bit two-level CLA adder are determined by:

$$C_{0,4,8,12} = G_{0,4,8,12} + BC_{-1,3,7,11} P_{0,4,8,12} \quad (6.12)$$

$$C_{1,5,9,13} = G_{1,5,9,13} + G_{0,4,8,12} P_{1,5,9,13} + BC_{-1,3,7,11} P_{0,4,8,12} P_{1,5,9,13}$$

$$C_{2,6,10,14} = G_{2,6,10,14} + G_{1,5,9,13} P_{2,6,10,14} + G_{0,4,8,12} P_{1,5,9,13} P_{2,6,10,14} + BC_{-1,3,7,11} P_{0,4,8,12} P_{1,5,9,13} P_{2,6,10,14}$$

The carry bits formed by equations (6.11) and (6.12) are combined in the summation unit in accordance with equation(6.5) to produce the resultant vector \vec{S}

For a one-level adder, a fanout of fifteen was experienced by the P_{14} output in developing C_{14} . Equation(6.12) shows that this fanout is reduced to three

for the two-level adder. The compromise is an increase in hardware complexity and four additional gate delays inherent in the two-level type.

Table(6.1) itemizes the order in which the intermediate results are formed, the unit in which they are formed, the amount of gate delay, and the pertinent equation numbers.

This section developed the theory of carry-look-ahead addition. The information and equations listed in Table(6.1) and Figure(6.1) provide the starting point for the VLSI project described in this thesis. Pipelining, carry-look-ahead addition, and PLA structures are combined in the next section to develop an initial "paper and pencil" functional unit design of the thesis project.

Intermediate Results	Unit	gate delay	Equation #
Pi's Gi's	CP/G	3	(6.3) & (6.4)
BPj's BGj's	BCLA	2	(6.9) & (6.10)
BCl's	CLA	2	(6.11)
Ci's	BCLA	2	(6.12)
Si's	SUM	3	(6.5)

Table(6.1)

3. Design Considerations

This section develops the initial block diagram of the VLSI project. Recall that CAD tools called "PLAGUE" and "PLAGEN" exist in the NPS VLSI design

inventory. They accept boolean equations as inputs and produce a CIF file as an output. Restrictions applicable to these two software tools are: forty inputs, forty outputs, and one hundred and fifty product terms. Capitalizing on the power and convenience of these two CAD tools, all of the combinatorial logic needed to realize a sixteen-bit adder is replaced by PLA structures. Since PLA structures are used exclusively to develop the equations whose numbers are given in Table(6.1), CLA addition is re-evaluated at a PLA level rather than a gate delay level. It is conceivable that on one extreme an adder can be made from one "large" PLA structure. The output \vec{S} could be tediously determined in terms of \vec{A} and \vec{B} . This produces a "single" stage adder. An adder of this type would have an enormous number of product terms. The huge fanout would probably render the design physically unrealizable using present technology. On the other extreme, the adder can be made from many smaller less complex PLA structures. Both extremes appear to have only two levels of gating when using the NOR-NOR form of PLA. This two levels of gating does not consider the fact that the complement of the input variable as well as the variable itself must be delivered to the input NOR plane. In order to supply the complement of the variable, an additional inverter must be placed between the input variable and the input plane. Also, for the NOR-NOR structure, an additional inverter must be utilized to invert each output term. This translates to a minimum gate delay of four for the NOR-NOR PLA structure. Four gate delays are experienced in a "complex" PLA as well as in a "simple" PLA. This might first suggest that the fastest adder is the extreme "single" stage model since both the complex and the simple versions have the same number of gate delays. This is not correct. The "number" of gate delays is equal but the "delay" in each level is a strong function of fanout. An input line (variable) is capable of charging one inverter gate above threshold in much less time than it takes to charge, for example, ten inverter gates above threshold.

This analogy applies to the one and two - level carry-look-ahead adders. Recall that a one-level adder has a total gate delay of eight while the two-level adder has a total of twelve. The maximum fanout for the one-level sixteen-bit adder was shown to be fifteen while the maximum fanout for the two-level adder (using four-bit BCLA units) was three. Therefore, it is possible for the two-level adder with the higher multiplicative gate delay factor to have a shorter overall add time. Clearly, the successful CLA - pipelined design requires a balance between the number of stages, the fanout in each stage, and the overall complexity of the design.

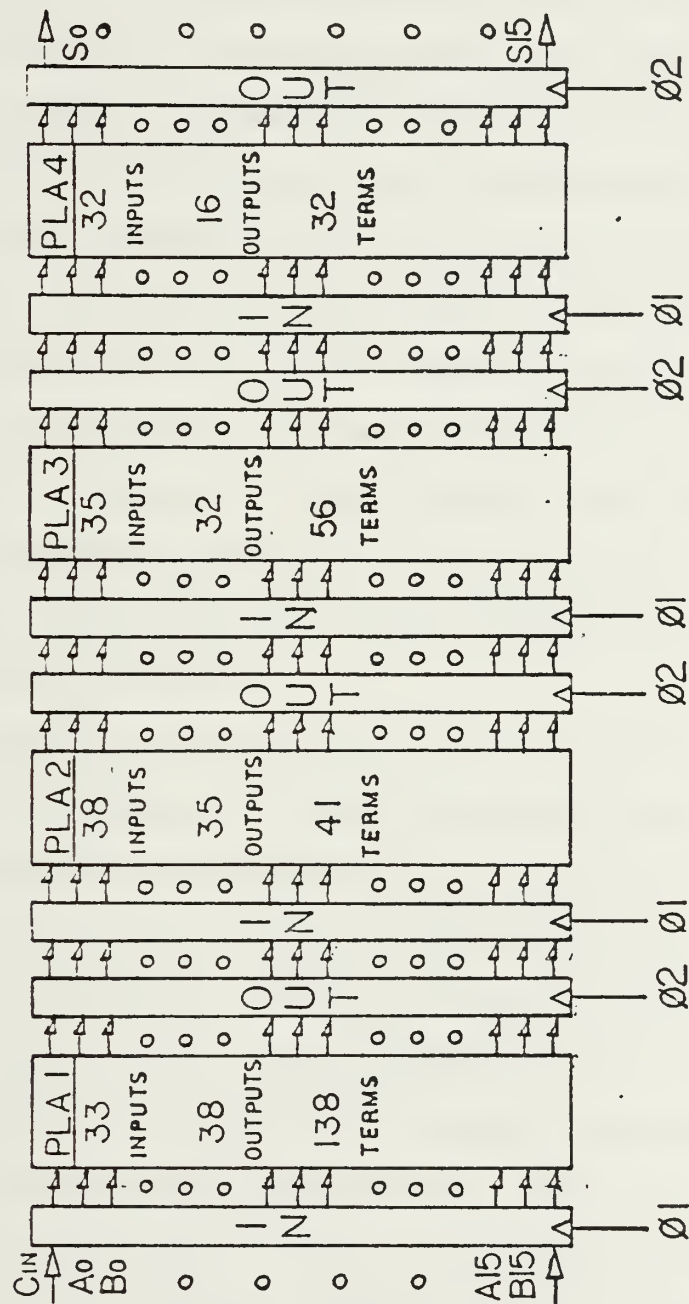
The first "attempt" at the design originated as a transformation of Table(6.1) into a pipelined adder. The table suggests five distinct stages. The first stage realized equations(6.3) and(6.4) in PLA form. The second PLA realized equations(6.9) and (6.10). PLA three implemented equation(6.11). Stage four developed equation(6.12). Finally, the last stage produced the final result by realizing equation(6.5). D type flip-flop circuits provided the interstage storage devices and were controlled by a two-phase non-overlapping clock. The "unused" variables were simply passed from input latch to output latch in synchronization with the data flow. [REF.1:chap.7] provides an excellent discussion on system timing in LSI circuits. Since no particular constraints were placed on the project other than those induced by the PLA CAD tools, an artificial one **was** created. The five-stage design was obvious from Table(6.1). The "requirement" for a four-stage adder was introduced to investigate the interaction between the number of stages,fanout,delay,and complexity. A re-design reduced the number of stages from five to four. This reduction was accomplished by recognizing that the block propagate and the block generate functions(equations 6.9 and 6.10) could be determined in terms of A_i and B_i and be included in the first PLA without exceeding the input,output,product term constraints. Complexity decreased but

the fanout in PLA #1 increased. An increase in fanout results in a decrease in operating speed. The tradeoff between complexity, speed, and fill time was considered acceptable for an initial VLSI design project. The exact equations and the input/output variables for each PLA are given in Appendix(E). This appendix contains the input files to the "PLAGUE" software tool. A (') symbol signifies a logical NOT, the (&) symbol signifies the logical AND and the (+) symbol is used for the logical OR. Two final simplifications were made to the design. First, not all of the variables are utilized in each PLA. The initial design passed these variables from input latch to output latch in synchronization with the data flow. It was determined through a comparison between allowable chip size and estimated PLA structure size that surface area on the chip was not a critical factor. As a result the "un-needed" variables in a particular stage were incorporated directly into the PLA structure. This was accomplished by defining the output variable as the input variable in the "PLAGUE" input file. This alleviated the need for "stray" latches and interconnecting wiring. PLA CAD tools automatically provided the extra circuitry. The second simplification was the replacement of the D type flip-flop latches by dynamic registers as described in [REF.1:p81]. This allowed the variables and their complements to be delivered to the input plane and the output from the output plane to undergo the necessary inversion. Figure (6.2) shows a block diagram of the VLSI design project.

As described in the following section, the next step in the design process is to verify that the logic of the design is functionally correct. This is done by emulating the design using the ILOGS software tool.

C. DESIGN VERIFICATION

ILOGS is an acronym for Interactive LOGic Simulator. This program utilizes models of the operating characteristics of metallic oxide semiconductor large



Figure(6.2) Block diagram

scale integrated circuits. Random access memories, read only memories, programmable logic arrays as well as conventional gates including inverters, AND, NAND, OR, NOR, XOR, etc. can be simulated using the ILOGS program. Clock specifications, input/output methods, power, and ground connections are also provided. The turn-on and turn-off delay times of the individual gates as well as the access time for the memory devices can also be assigned in order to more closely realize the real world design. Indigenous to most digital logic circuits is a large number of identical logic structures. ILOGS contains a macro "definition" feature so that these identical structures need only be defined once on a primitive level. Subsequent usage of the structure can be accomplished by "expanding" the macro definition. New node names are assigned in the expansion statement. The result is a replication of the initial definition. From the above discussion, it should be obvious that ILOGS is a very complex and powerful software tool.

It is conceivable that a small computer could be emulated by the ILOGS program. Because of the complexity of ILOGS, it is beyond the scope of this thesis to provide an in-depth discussion on all of the data structures and terminal commands found in ILOGS. Rather, only the necessary information concerning the verification of the VLSI project is provided. The *ILOGS USER'S MANUAL* version 2H [REF. 10] should be referred to by first time users of ILOGS.

All of the software tools discussed up to this point are processed on the VAX 11-780 computer under the UNIX operating system. ILOGS is run under the VMS operating system. NPS possesses two VAX 11-780 computers. One uses the UNIX operating system and one uses VMS. Fortunately, the VAX computer which utilizes the VMS operating system is also capable of emulating the UNIX operating system and the "vi" editor. Therefore, it is not necessary to learn an additional operating system and editor to use the ILOGS program. Although it is culturally

enriching to become familiar with two different editors and operating systems, it is recommended for a person who is familiar with neither VAX system to learn and use the UNIX/"vi"editor on both computers. Appendix(A) describes the VAX/UNIX system. By learning one system well,the overall proficiency of the designer is increased. The necessary information to use UNIX on the VMS VAX computer is given in the login sequence.

There are two types of information which ILOGS processes. They are data information and terminal commands. Data information can be subdivided into three main categories. They are: 1.clock/table data, 2.network data, and 3.output specifications. The clock data is used to define highly repetitive timing information as well as constants such as VDD and GRD. The table data provides a means to generate inputs to the design if needed. Network data is the "heart" of the digital circuit. This data describes the topology and characteristics of the circuit to be simulated. Included are basic gates,ROM's RAM's,PLA's,macro definitions,macro expansions and connections through the use of identical node names. Two nodes at different points in a circuit are considered to be connected if they have the same names. Output specifications designate the nodes that the user desires to analyze. Simulation of a circuit begins with the creation of a file. This file contains all the necessary data information to simulate the design. Once this file is completed, terminal commands are used to perform the desired operations on the data file. Several of the more frequently used terminal commands are explained later in this chapter.

The translation of Figure (6.2) into a data file is described in the following discussion. The actual data file for the sixteen-bit adder required 953 lines of code. Only parts of this data file are included in this thesis since the structure of the design is highly repetitive. The only difference between stages is the size and function of the PLA's.The input/output registers are identical in structure but

differ in length. A good understanding of the operation of ILOGS can be obtained by analyzing the three main groups of data information as they appear in the adder file. Figure(6.3) lists lines (7-25) of the data information file for the sixteen-bit adder. This section of the file describes the clock operations and input vectors utilized by the circuit. Comment statements are denoted by a "\$" sign. On any line, all characters to the right of a "\$" are considered to be comments and are disregarded by ILOGS. Lines(9 and 10) describe VDD and GROUND. A clock statement begins with a "name" and is followed by ".CLK" to denote that the node "name" is a clock and that "time-state" pairs follow. Line(9) is interpreted as : a node named VDD is a clock and at time zero VDD is "driven" to a logic "1" (D1) and remains high indefinitely. This is simply a method of providing

```

7  $ define clocks
8  $
9  VDD .CLK 0 D1
10 GRD .CLK 0 D0
11 PHI1 .CLK 0 D1 20 D0 50 D1 70 D0 100 D1 120 D0 150 D1 170 D0 200 D1 220
12 + D0 250 D1 270 D0 300 D1 320 D0
13 PHI2 .CLK 0 D0 25 D1 45 D0 75 D1 95 D0 125 D1 145 D0 175 D1 195 D0 225
14 + D1 245 D0 275 D1 295 D0 325 D1 345 D0
15 $
16 $ define inputs
17 $
18 .TABLE IA15 IA14 IA13 IA12 IA11 IA10 IA9 IA8 IA7 IA6 IA5 IA4 IA3 IA2 IA1 IA0
19 + IB15 IB14 IB13 IB12 IB11 IB10 IB9 IB8 IB7 IB6 IB5 IB4 IB3 IB2 IB1 IB0 IC-1
20 0      0110101010011100  1110010101100011 1
21 35     1111111000111000  0101110100000100 0
22 85     1000101000000100  0110101111100000 0
23 135    0100111111010100  0010101010101010 0
24 .EOT
25 $

```

Figure (6.3) Clock and vector information

the necessary power connection to the circuit. Lines(11-14) describe the two-phase non-overlapping clock. The node named PHI1 is driven to "1" at time zero and remains at "1" until time 20. At time 20 PHI1 is driven to "0" (D0) and

remains there until time 50 etc.etc. PHI2 is described in a similar manner. When longer repetitive clock sequences are needed, a "repeat" feature is utilized. This allows the clock to run indefinitely. Input data is shown in lines(18-24). The ".TABLE" on line (18) and ".EOT" (end of table) on line (24) delineate the extent of the table. The input variables starting with "IA15" and ending with "IC-1" (a "+" indicates a continuation of the previous line) assume the values of the binary digits listed on line(20) from time 0 to time 35⁻, on line(21) from time 35 to time 85⁻ etc. For example, at time 135, IA15=0, IA14=1, IA13=0,...IB1=1, IB0=0 and IC-1=0.

Figure (6.4) lists lines (27-37). A macro definition is described by lines (28-33). A macro whose name is PLAT begins on line (28) and has four external nodes called (IN,PHI1,OUT,OUT-). Lines (29-32) define the circuitry of PLAT. This circuit uses three NMOS inverter gates denoted by the gate type designator (.INV/N). The name of the output node of any conventional gate is the name of

```

27 $
28 .MACRO PLAT IN PHI1 OUT OUT-
29 OUT .INV/N A2
30 A2 .INV/N A1
31 OUT- .INV/N A1
32 A1 IN .SWCR PHI1
33 .EOM PLAT
34 *LAT0 PLAT IA0 PHI1 A0 A0-
35 *LAT1 PLAT IA1 PHI1 A1 A1-
36 *LAT2 PLAT IA2 PHI1 A2 A2-
37 *LAT3 PLAT IA3 PHI1 A3 A3-

```

Figure (6.4) Macro definition and expansion

that gate and is always listed first. In this case, (OUT,A2,OUT-) are the three inverter gates. Inputs to conventional gates are listed after the gate type designator and are (A2,A1,and A1). Line (32) describes a "switchable-wired-

OR"(.SWOR) circuit. ILOGS version 2H does not support MOS "pass" transistors. Therefore, the NMOS pass transistors in the input/output registers had to be modeled as a switchable-wired-OR. Both possess nearly the same characteristics. For the NMOS pass transistor a positive voltage above threshold on the gate effectively shorts the source and drain terminals. When the voltage on the gate is below threshold, the short is replaced by a very high impedance. The (.SWOR) circuit operates in the same manner. In line(32), the node names listed before (.SWOR) become connected when the enable signal-the node name following (.SWOR)-becomes "high" and disconnected when the enable signal becomes "low". In this case, nodes (A1) and (IN) are connected when PHI1 is "high" and disconnected when PHI1 is "low". The function of PLAT is to deliver a single variable and it's complement to the input of the PLA. The (.SWOR) allows the charge to be "trapped" by disconnecting (A1) and (IN). Figure (6.5) shows a schematic diagram of this macro definition. Lines(34-37) of Figure(6.4) show four macro

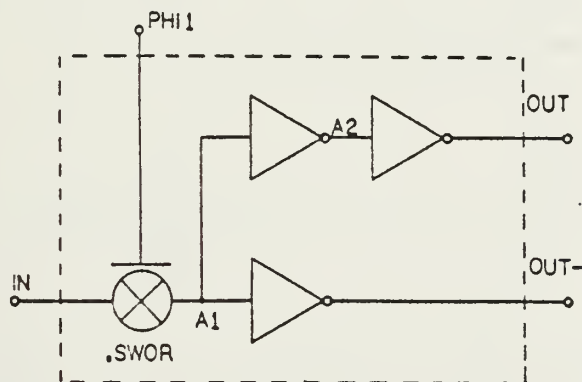


Figure (6.5) Circuit diagram of macro PLAT

expansions of the macro definition named PLAT. Thirty-three macro expansions of PLAT are needed to realize the dynamic input register that provides inputs to the first stage PLA. For example, in line (36), *LAT2 assigns a new name to the

macro PLAT. It is now known to ILOGS as LAT2 with external connections(IA2,PHI1,A2,A2-). These newly defined external connections must be in the same order as (IN,PHI1,OUT,OUT-). The replacement name (IN) now becomes (IA2), PHI1 remains unchanged,(A2) becomes (OUT), and (A2-) replaces (OUT-). The replacement names must match the order of specification for the macro definition. Line (36) of Figure(6.4) realizes the following circuit shown in Figure (6.6). A comparison between Figures (6.6) and (6.5) shows the relation between a macro definition and a macro expansion. This is a valid expansion even though (A2) appears on both the input and output of the same inverter circuit in Figure (6.6). ILOGS separates node names used inside a macro definition from those referenced outside of a macro definition. To realize this single input circuit, only one line of code was needed instead of four. There are one-hundred

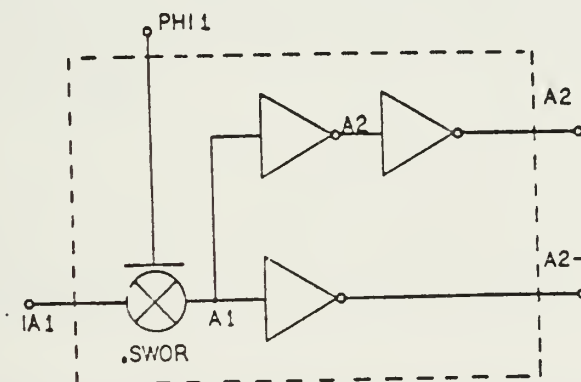


Figure (6.6) An expansion of the macro PLAT

thirty-five PLA input circuits needed to realize the four-stage adder. Four-hundred and five lines of code were saved by using the macro expansion feature of ILOGS. A similar procedure was used for the PLA output circuitry. A single macro named "STAGEOUT" was defined. An additional one hundred twenty one

expansions of STAGEOUT were utilized. The power of the macro feature in ILOGS network data is obvious.

Lines (855-872) are shown in Figure (6.7). Line (855) determines the beginning of stage four PLA and ".EOP"(not shown) signifies the end. The input plane and the output plane both implement the NOR function. This is shown by (.NOR/N .NOR/N) on line (855). The input variables are listed starting with "CARRY-1" in line (856). The complement of a variable is denoted by a trailing (-) variable string from the output variable string. Output variables begin on line(863) with "SUM0" and end with "SUM15" followed by a slash. A one-to-one

```

855 .PLA .NOR/N .NOR/N
856 + CARRY>1 CARRY>1~ CARRY0 CARRY0~ CARRY1 CARRY1~ CARRY2 CARRY2~
857 + CARRY3 CARRY3~ CARRY4 CARRY4~ CARRY5 CARRY5~ CARRY6 CARRY6~
858 + CARRY7 CARRY7~ CARRY8 CARRY8~ CARRY9 CARRY9~ CARRY10 CARRY10~
859 + CARRY11 CARRY11~ CARRY12 CARRY12~ CARRY13 CARRY13~ CARRY14 CARRY14~
860 + PF00 PF00~ PF10 PF10~ PF20 PF20~ PF30 PF30~ PF40 PF40~ PF50 PF50~
861 + PF60 PF60~ PF70 PF70~ PF80 PF80~ PF90 PF90~ PF100 PF100~ PF110 PF110~
862 + PF120 PF120~ PF130 PF130~ PF140 PF140~ PF150 PF150~/
863 + SUM0 SUM1 SUM2 SUM3 SUM4 SUM5 SUM6 SUM7 SUM8 SUM9 SUM10 SUM11 SUM12
864 + SUM13 SUM14 SUM15/
865 +U1 X.....X.....
866 + X.....
867 +U2 .X.....X.....
868 + X.....
869 +U3 ..X.....X.....
870 + .X.....
871 +U4 ...X.....X.....
872 + .X.....

```

Figure (6.7) Partial description of stage four PLA

positional linear relationship holds between the location of the input/output variables in the list and the following array connection terms. An (X) represents a connection and a (.) represents no connection. Consider lines(865 and 866). Line(866) is a continuation of line(865). The sixty-four input variables correspond (one-to-one) with the sixty-four possible connections on line(865).

Similarly, the sixteen output variables correspond to the sixteen possible connections on line(866). For example, the first (X) in line(865) corresponds to the first input variable $CARRY-1$. The second (X) in the same line(position thirty-four) corresponds to $\overline{PF00}$. Both of these input variable are related to $SUM0$ since the (X) connection appears in position one of line(866). In the same manner, lines(867 and 868) relate the variables $\overline{CARRY-1}$ and $PF00$ to $SUM0$. The ILOGS NOR-NOR PLA realization translates lines(865 through 868) to the following:

$$SUM0 = \overline{U1} + \overline{U2}$$

$$SUM0 = \overline{(\overline{CARRY-1} + (\overline{PF00}) + (\overline{CARRY-1}) + (\overline{PF00}))}$$

The application of DeMorgan's theorem allows $SUM0$ to be written as:

$$SUM0 = \overline{(\overline{CARRY-1})(PF00) + (\overline{CARRY-1})(\overline{PF00})}$$

The $SUM0$ variable is inverted by the expansion of the macro STAGEOUT. This operation produces the NOT of $SUM0$ and is denoted $FS0$. $FS0$ (final sum 0) then has the following equation.

$$FS0 = (\overline{CARRY-1})(PF00) + (\overline{CARRY-1})(\overline{PF00})$$

This is the required XOR logical function needed to produce the final sum bit (see equation 6.5). PLA structures and the associated input/output circuitry provide all of the network data to realize the sixteen-bit adder in ILOGS readable code.

The (.OUTPUT) specification is a means to observe the operation of the circuit. Lines (948-953) shown in Figure (6.8) provide a concrete example. Node names following (.OUT) are the nodes of interest to the designer. States of each node listed are included in the output table. In this case, the two-phase non-overlapping clock and the sixteen sum bits are the nodes of interest. Terminal commands can display or store the output table in various forms.

Although only portions of the sixteen-bit adder file in ILOGS readable code are shown, enough information has been supplied to understand the transformation of the adder from block diagram form (Figure 6.2) to design verification form.

```

948 $ output the sums
949 $
950 $
951 .OUT PHI1;;PHI2;;FS15;FS14;FS13;FS12;FS11;FS10;FS9;FS8;FS7;FS6;FS5;
952 +   FS4;FS3;FS2;FS1;FS0
953 END

```

Figure (6.8) Output specifications

Many terminal commands exist in ILOGS. Only those necessary to verify the proper operation of the adder are discussed in this thesis. Once the design file has been created and is residing on bulk storage(disk), a command to invoke ILOGS is issued. Under the VMS operating system "RUN ILOGS" accomplishes this task. The cue "ENTER COMMAND" is returned. Retrieval of the design file from disk is the next step. "INPUT [FILENAME]" reads the file from the disk. "SIMULATE FROM t1 TO t2" invokes ILOGS to simulate the design from time (1) to time (2). The starting and stopping times should be consistent with the clock specifications. Simulation time frames can be less than but not greater than the clock duration. When simulation is completed, the command "(PRINT,TYPE,or

STORE) OUTPUT FROM t1 TO t2 ON CHANGE" causes ILOGS to print on the lineprinter, type on the terminal screen, or store on disk the states in tabular form of all the nodes listed in the (.OUTPUT) specifications each time any node listed in the output specifications changes state. Many more commands exist and there are many options, however, the above commands provide the basic repertoire needed to verify the design. When the simulation is completed and results recorded, "EXIT" will return control to the VMS operating system.

Figure (6.9) is the ILOGS output that verified the adder design. Note that an output is not obtained until the fourth time PHI2 is asserted "high". Also, a different correct sum is displayed for each subsequent assertion of PHI2. The four resultant sums are derived from the input vectors listed in the TABLE specifications (lines 18-24).

With the design verified for proper operation, the next step in the VLSI design procedure is to begin a "bottom-up" layout of the project utilizing the chip layout language (CLL).

D. LAYOUT

Chapter four gives an in-depth description of the CLL CAD tool. The building block approach, PLA generation, and the chip layout language are utilized in this section to produce the VLSI design. Each step of the layout is discussed with reference to the file or program that was written to realize the final chip. For clarity, all of the files or programs are listed in order of reference in Appendix(E).

The Stanford University cell library is used exclusively in this design. Four major "cells" were required that were not in the cell library. As a result, the missing cells were created and added to the list of useable "puzzle" pieces. The first step in this design was the creation of the four PLA structures. PLA's one

VLSI I

	P	P	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	H	H	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	I	I	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0	
	1	2	5	4	3	2	1	0											
TIME																			
0	1	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
20	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
25	0	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
45	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
50	1	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
70	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
75	0	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
95	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
100	1	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
120	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
125	0	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
145	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
150	1	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
170	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
175	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
195	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
200	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
220	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
225	0	1	0	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0
245	0	0	0	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0
250	1	0	0	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0
270	0	0	0	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0
275	0	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1	0	0	0
295	0	0	1	1	1	1	0	1	0	1	1	1	1	0	0	1	0	0	0
300	1	0	1	1	1	1	0	1	0	1	1	1	1	0	0	1	0	0	0
320	0	0	1	1	1	1	0	1	0	1	1	1	1	0	0	1	0	0	0
325	0	1	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	0	0
345	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	0	0

Figure (6.9) ILOGS design verification

through four listed in Appendix(E) describe the input files that were submitted to the CAD tool "PLAGUE". File "pla1"(CIF#950) lists the boolean equations that realize the P_i 's, G_i 's, BP_j 's, and BG_j 's in terms of the A_i 's and B_i 's. File "pla2"(CIF#951) forms the BC_i 's. File "pla3" (CIF# 952) develops the C_i 's and file "pla4"(CIF#953) produces the S_i 's. There are several options available that can effect the output of the PLAGEN program. The PLAGUE-PLAGEN tools were initially used without any options to obtain the size of the individual structures. Using the sizes of the PLA's and of the selected input/output circuitry, a floor plan was created, Figure(6.10), that accommodated the chip size limitations of $6890 \times 6300 \mu\text{m}$. Standard cells for PLA input /output circuitry were selected from the Stanford cell library. PlaClockIn and Afterburner were used for the input and PlaClockOut was used for the output. The input and output circuits have the capability of either being attached to the bottom or top of the appropriate plane. It is possible to erroneously transpose the input variables and their complements. The PlaClockIn/Afterburner combination accepts a variable from the bottom (arbitrary reference) and, after inversion and buffering, outputs the variable on the right top and the complement of the variable on the left top for insertion into the PLA plane. If this cell combination is rotated 180° , (input from the top) then the variables are switched. Care must be exercised when attaching the input circuitry to the PLA plane. The "-c" option of PLAGEN automatically complements the connections within the PLA plane. Since the output plane only has one line per output variable, this does not apply to the output circuitry. However, PLAGEN automatically provides PlaPullup pairs on the top of the output PLA plane. If layout constraints require that the output of a PLA must be taken from the top, then the "-o" option must be used. This option prevents the occurrence of the PlaPullups on the output plane. PlaPullups deleted by the "-o" option must be replaced at the opposite end of the output

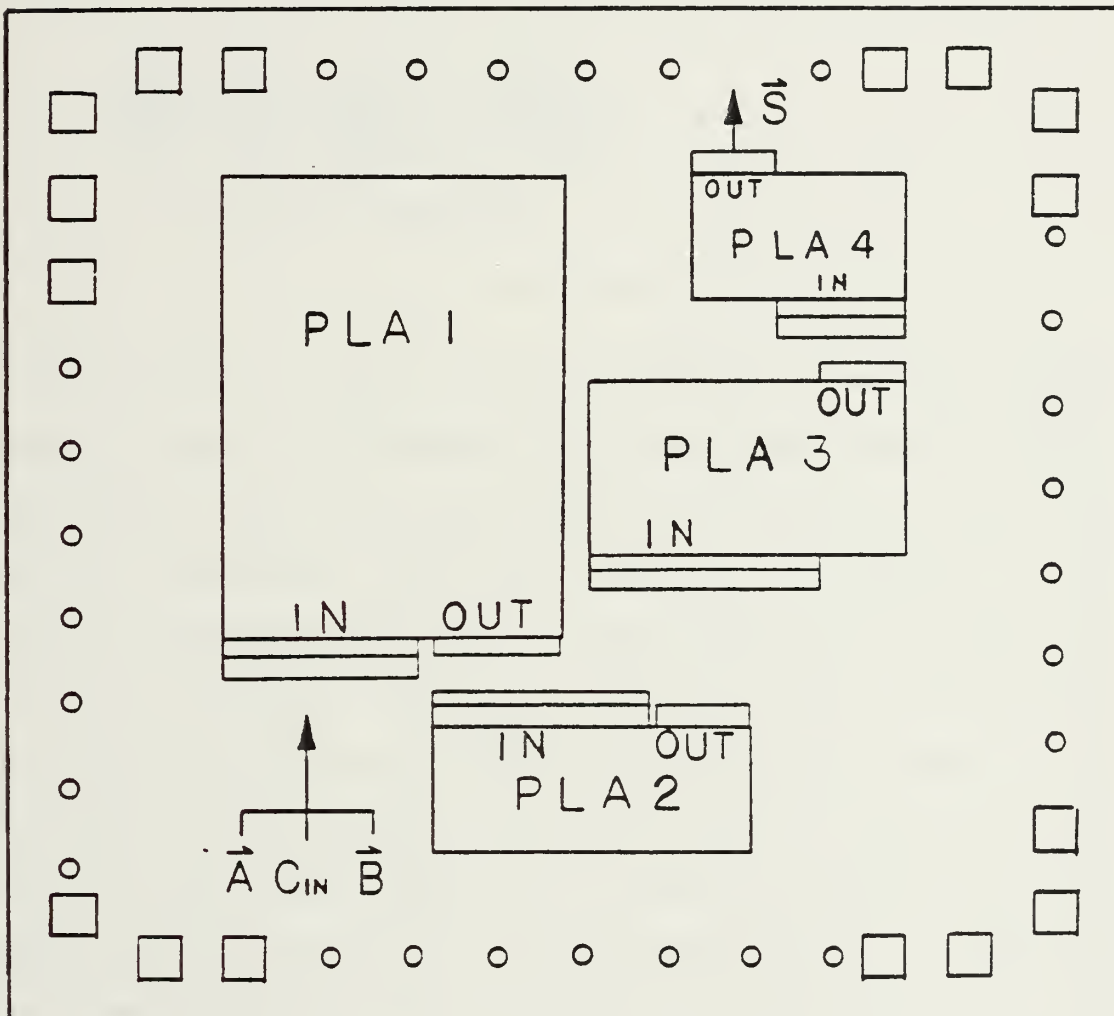


Figure (6.10) Floor plan

PLA plane for proper operation. The following four commands were executed to develop the PLA structure consistent with the floor plan. The results of these four commands are the addition of four new cells that can be utilized as any other cells in the cell library.

```
plague <pla1 | plagen -c > pla1.cif
plague <pla2 | plagen -o > pla2.cif
plague <pla3 | plagen -c -o > pla3.cif
plague <pla4 | plagen -o > pla4.cif
```

PLA GENERATION COMMANDS

The next step in the design process is to attach the necessary input/output circuitry and replace any PlaPullups that may have been deleted by the "-o" option. Program "stage1.cll" in Appendix(E) provides a concrete example of how this task is accomplished. This program attaches the cells PlaClockIn, Afterburner, and PlaClockOut to the PLA structure #1. Lines (2 and 4) of "stage1.cll" allow for the cell library and the newly created PLA#1 to be used by this program. The portion of the program between the brackets can be translated as follows. Line(8) "gets" pla1 and places the lower left corner of its bounding box at x=0,y=123. Line(10) "gets" Afterburner and places the lower left corner of its bounding box at x=16,y=58. Then line(9) causes this cell to be repeated 33 times in the x direction with only one occurrence in the y direction. Remaining lines are interpreted in the same manner with different cells and starting points. If any transformations were listed, they would have been executed before bounding box placement and repetition took place. Various starting points are determined by measuring the center coordinates of the input/output lines of the cells to be attached. The coordinate for the lower left corner of the bounding

box can then be determined. When PlaClockIn has its lower left corner located at $x=15$ $y=0$ as shown in line(12) of "stage1.cll", the two output lines abut precisely to the two input lines of Afterburner provided that Afterburner has its' lower left corner at $x=16$, $y=58$. When the program is executed, another cell is formed. It consists of the PLA structure with all of the input/output circuitry attached. The lower left corner of the bounding box has its coordinates at $x=0$ $y=0$. Three additional PLA stages are created in the same manner.

With the four main stages completed, the next step is to layout the input/output bonding pads. This is accomplished in program "stage5.cll". The number of bonding pads was determined to be fifty-three. This included thirty-two inputs for \vec{A} and \vec{B} , one input for BC_{-1} , sixteen outputs for \vec{S} , two inputs for PHI1 and PHI2, and two for VDD and GROUND. To alleviate excessive wire run length and "cross-over" complexities, the input pads were distributed as close as possible to the input area of stage one. Similarly, the output pads were placed as close as possible to the output of stage four. The execution of this program produced a "cell" of dimension 2500λ by 2700λ . Fifty-three bonding pads are located around the outer edge with a large void in the middle.

A final program is needed to complete the design. It must combine the five stages into one then provide the interconnecting wiring. This program is called "tot.cll". Stage five has the lower left corner of its bounding box located at $x=0$ $y=0$. The remaining stages are strategically placed within stage five to allow enough room for wire runs between stages and bonding pads. The x-y coordinates for stages 1-4 can be seen in lines 11-14 of program "tot.cll". Interconnecting wiring to complete the chip is provided by the "wire" statements in the remaining lines of "tot.cll". The names of the designers were added in the polysilicon level by "including" the program "designer.cll" in "tot.cll". Execution of "tot.cll" produced a CIF file that contained all of the necessary elementary

rectangles on the proper levels to realize the adder. This design was subjected to two remaining tests before submitting it for fabrication. A design rule check and a logic simulation are the next steps in the design process.

E. DRC

A "tot.sco" file was created from the "tot.cll" program. The DRC uses this file to search for design rule violations. The check took several hours to complete. It returned a file with seven errors. These were quickly found by using a plot of the chip and the coordinates listed in the error file. Corrections were made to the wire list and the chip was again submitted to the DRC. The second run was completed error free.

F. SIMULATION

The circuit extractor provides a means to identify various nodes in the design by number. Nodes of interest (input pads, output pads, VDD, and GROUND) are each assigned a label in order for the chip to be simulated. For example, locate line(32) of file "final.sym" in Appendix(E). The output plot derived from the extractor has numbers associated with many nodes. In this case, #11301 defines the input bonding pad that the designer called A13. The "final.sym" file is created to prescribe this labeling for all nodes of interest. The labels are then used by the event driven simulator (esim). Chapter four, section D, describes the mechanics of the event driven simulator. The file "sim.in" in Appendix(E) prescribes the clock(K), the labeled nodes of interest to "watch"(W), and the high(h) and low(l) input nodes. As a result of the circuit simulation, the file "sim.out" was produced. The values for the inputs and outputs are listed for each cycle. As expected, the first three cycles produced no output (OUT=XXX...X) but on the fourth cycle the correct sum was obtained. Decimal output on

line(28) <35294> is the sum of line(12) CIN=1, line(14) B=23270, and line(15) A=12023. These values occurred as inputs three cycles earlier in the sequence. Several simulations were completed using various values for the input vectors. All cases produced the correct output. Since the design was made entirely from the cell library or computer generated cells a static check was not needed.

The design passed ILOGS verification, a design rule check, and an event driven simulation. It was then considered ready for fabrication.

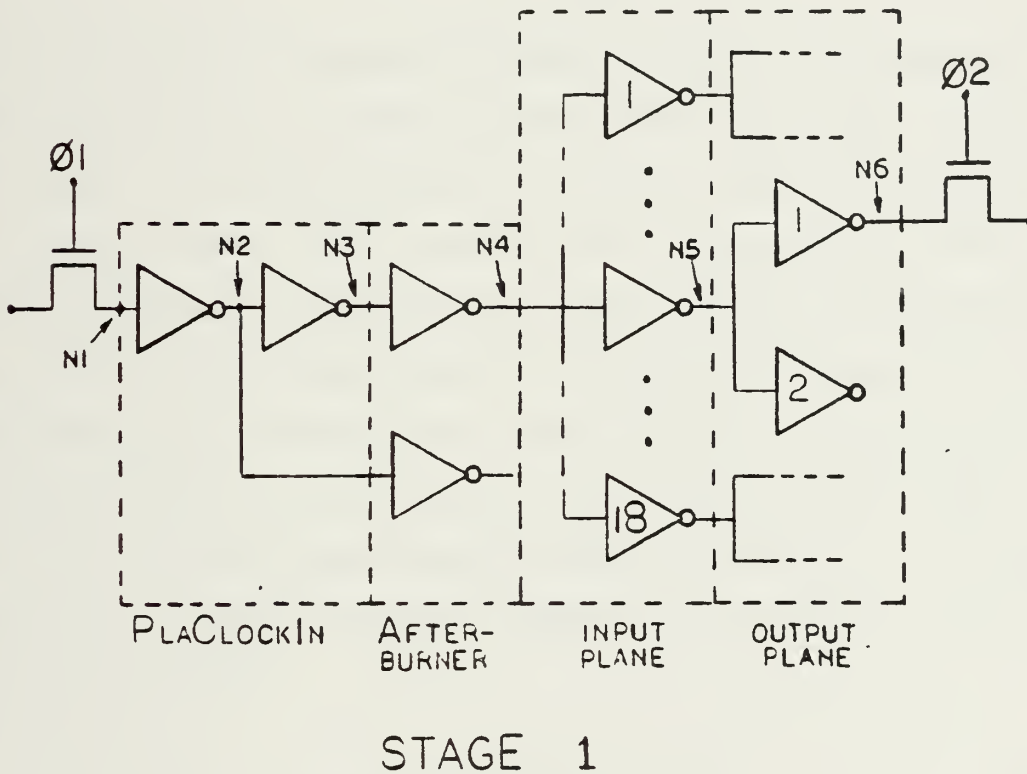
VII TESTING

A. EXPECTATIONS

The design was intended to add two sixteen-bit vectors and a carry-input bit to produce the sixteen-bit sum of the inputs without a carry-output bit(overflow). To produce the carry-out bit, it would be necessary to implement additional block propagate and block generate functions. This cannot be accomplished by a minor modification of the existing design because the input and product term limitations for PLA# 1 are exceeded. The absence of the overflow bit is not considered to be a significant degradation. Since the adder obtains its inputs from a analog-to-digital converter, the analog voltage input could be properly limited and scaled to prevent bit weights that would cause an overflow. The lack of an overflow or carry- out bit would, however, prevent the combination of two sixteen-bit adders into one thirty-two-bit adder.

The design was also intended to generate the sums at a very fast rate. It was discussed in chapters two and six that the fastest clock rate at which the adder will operate depends on the slowest stage. The slowest stage is that with the largest fanout. PLA# 1 determines the clock rate for this design. Mead and Conway [REF.1:sections.1.3,1.5,1.13] provide some insight into the very complex topic of system timing analysis. To perform more than a worst case timing analysis, requires an in-depth discussion of device physics and electrical parameters which is beyond the scope of this thesis. An estimate of the operating speed was obtained by using the guidelines cited in the aforementioned sections of [REF.1]. Shown in Figure(7.1) is an abstract representation of the "worst" case conditions for stage one of the adder. A maximum fanout of eighteen exists in the input PLA plane and a maximum fanout of two exists in the output PLA plane. When a series of inverters is cascaded as in Figure(7.1), and a change of

input voltage occurs, the charge from "high" nodes is removed through switched-on pull-down transistors. Additionally, the "low" nodes are charged



Figure(7.1) "Worst" case abstraction of Stage one

by the previous pull-up transistors. The amount of time for a pull-down transistor to "sink" charge is less than that for a pull-up transistor to supply charge.

Let the time required for a pull-down transistor to remove charge from a node equal (τ) . Then, the time for a pull-up transistor to supply charge to a single follow-on gate is $(k\tau)$ where (k) is equal to the ratio:

$$k = \frac{Z_{pu}}{Z_{pd}} = \frac{\frac{L_{pu}}{W_{pu}}}{\frac{L_{pd}}{W_{pd}}} \quad (7.1)$$

Here (Z) is equal to the length (L) to width (W) ratio of the gate region and "pu" denotes pull-up and "pd" denotes pull-down. When fanout occurs, the time to sink charge from (f) nodes becomes $(f\tau)$ whereas, the time to supply charge to (f) nodes becomes $(kf\tau)$. Since it requires more time for a node to be charged, the worst case occurs when the "Afterburner" cell is tasked with charging up the inputs to all eighteen inverters. This occurs when a logic "0"(0 volts) input follows a logic "1"(5 volts) input. Assume a logic "1" has been clocked in on $\phi 1$ and has stabilized all the nodes from node one(N1) to node six(N6) in Figure(7.1). The graphical analysis shown in Figure(7.2) assumes that the Afterburner cell is a simple inverter with $(k=8)$. The total time for the logic "0" input to stabilize N6 is:

$$t_{\min} = \tau + k\tau + \tau + f_1 k\tau + f_2 \tau \quad (7.2a)$$

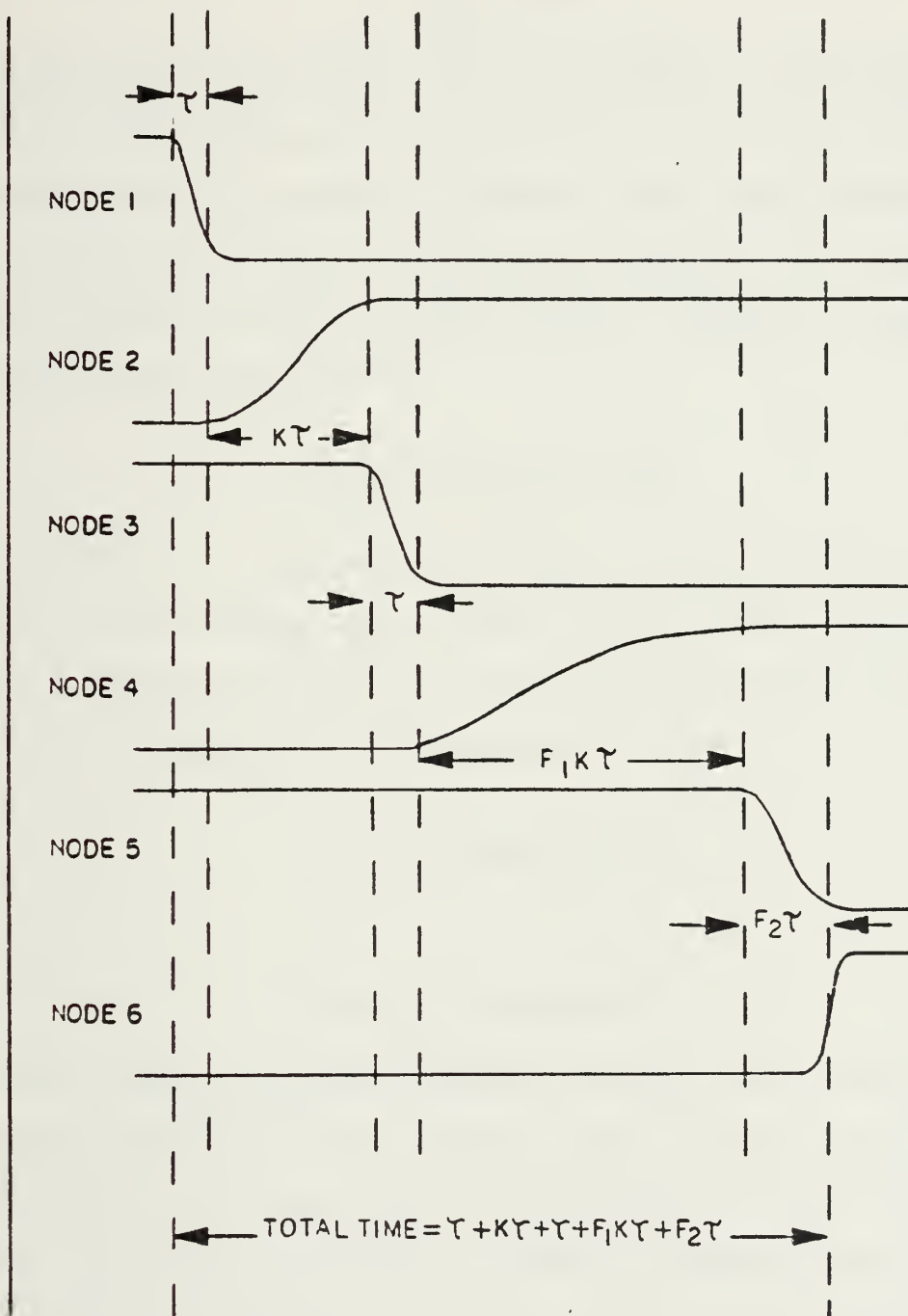
In this case

$$k=8 ; f_1=18 ; f_2=2$$

So

$$t_{\min} = \tau + 8\tau + \tau + 144\tau + 2\tau \quad (7.2b)$$

Since the Afterburner cell is a superbuffers and has approximately four times the current sourcing capability of a standard inverter, the fourth term in



Figure(7.2) Graphical timing analysis for the "worst" case.

equation(7.2b) can be reduced by a factor of four. This gives:

$$t_{\min} = 48\tau \quad (7.3)$$

The value of (τ) is approximately equal to six-tenths of a nanosecond. Tau is obtained from the fabricator's specification sheet received with each set of chips. The value of t_{\min} is equal to the total time for a single clock phase (φ_1) if stray capacitance is ignored. Normally, stray capacitance is at least as great as the capacitance found in the gate circuitry. Therefore, a conservative approach is to double (t_{\min}). Thus:

$$t_{\min} = 60 \text{ nanoseconds} \quad (7.4)$$

Since t_{\min} is the total time for one clock phase of a two-phase non-overlapping symmetrical clock scheme, this value must again be doubled. Also, a finite amount of time must be allotted for the non-overlapping portion of the clock. This adds approximately another five nanoseconds. Finally:

$$t_{\min} \approx 125 \text{ nanoseconds} \quad (7.5a)$$

and

$$freq_{\max} \approx 8 \text{ megahertz} \quad (7.5b)$$

Equation(7.5) shows the expected values for the adder when a two-phase non-overlapping symmetrical clock is used. By using a two-phase non overlapping asymmetrical clock, $freq_{\max}$ can be increased. Phase one (φ_1) of the clock must be long enough to allow PLA# 1 to function properly, but phase two (φ_2) may be shortened considerably since there is only one inverter stage between the phase two pass transistor and the next phase one pass transistor. Symmetrical clock schemes are much easier to implement than asymmetrical clock

schemes. Thus, a speed versus complexity tradeoff in terms of hardware and synchronization is apparent.

B. PROCEDURES

MOSIS requires that all CIF files be transmitted over the ARPANET. Since the VAX computer at the Naval Postgraduate School does not yet have this capability, the design was taken to Stanford University on magnetic tape and was submitted for manufacture by the Stanford Electronics Labs(SEL). The completed chip was returned to SEL approximately eight weeks later. SEL graciously permitted our use of their IC testing equipment to test the chips. This alleviated the need to design and build a custom made tester which saved an enormous amount of time.

The tester(a custom made design soon to be available to the public) interfaces the chip under test to a computer. A test program must be written in the "C" programming language for submission to the source program called "MINT". This test program is very similar to the file used to simulate the chip under "ESIM". The test program causes prescribed high and low voltages to appear at program defined input pins at prescribed times. Output pins and the expected values at the output pins are also prescribed in the test program. The computer then provides appropriate cues to the user if the expected values do not agree with the actual values. A plan for testing was created. First a short program was written to perform a perfunctory test of the chip. This program can be seen in Figure(7.3).The "define" statements in lines(1-6) tell the computer which pin numbers correspond to named nodes. For example, in line(18), the A vector is equal to all zeros. Line(19) shows the B vector equal to (0111...1) and the carry bit (C) set to one. "CLK" forces the computer to cycle through the steps defined in line(7). The non-overlapping feature is automatically supplied by the

INITIAL TEST PROGRAM

```
1  #define A 53 51 49 47 45 43 39 37 35 33 31 29 27 23 21 19
2  #define B 54 52 50 48 46 44 42 38 36 34 32 30 28 26 22 20
3  #define C 55
4  #define PHI1 18
5  #define PHI2 17
6  #define S 61 62 63 64 2 3 4 5 6 10 11 12 13 14 15 16
7  #define CLK PHI1 = 1;PHI1 = 0;PHI2 = 1;PHI2 = 0;

8  PHI1 = 0; PHI2 = 0;
9  A = 0 1 1 0 1 0 1 0 1 0 0 1 1 1 0 0;
10 B = 1 1 1 0 0 1 0 1 0 1 1 0 0 0 1 1; C = 1;
11 CLK;
12 A = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
13 B = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; C = 0;
14 CLK;
15 A = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;
16 B = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; C = 0;
17 CLK;
18 A = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
19 B = 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1; C = 1;
20 CLK;
21 S? 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0;
22 A = 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0;
23 B = 0 1 0 1 1 1 0 1 0 0 0 0 0 1 0 0; C = 0;
24 CLK;
25 S? 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
26 A = 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0;
27 B = 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 0; C = 0;
28 CLK;
29 S? 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;
30 A = 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0;
31 B = 0 1 1 0 1 0 1 1 1 1 1 1 0 0 0 0; C = 0;
32 CLK;
33 S? 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
34 A = 0 1 0 0 1 1 1 1 1 1 0 1 0 1 0 0;
35 B = 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0; C = 0;
36 CLK;
37 S? 0 1 0 1 1 0 1 1 0 0 1 1 1 1 0 0;
38 A = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
39 B = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; C = 0;
40 CLK;
41 S? 1 1 1 1 0 1 0 1 1 1 1 0 0 1 0 0;
42 CLK;
43 S? 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 0;
```

Figure(7.3) Initial test program

computer. Line(21)--(S ? 0101000000000000)-- indicates to the computer the values expected on the output pins whose numbers are defined in line(6). If these values differ from the actual values, a series of cues are printed out to the user's terminal. A sample is shown in Figure(7.4). The second step in the test plan was to write a more complex program that would supply all combinations of test vectors to the inputs of the chip. The absence of any cues on the user's terminal would indicate a thorough and successful test of the chip. Unfortunately, the second step never had to be implemented.

ERROR CUES

```
1  "/chip.test", line 42: pin 14 should be 0
2  "/chip.test", line 42: pin 15 should be 0
3  "/chip.test", line 57: pin 15 should be 0
4  "/chip.test", line 57: pin 16 should be 0
5  "/chip.test", line 62: pin 62 should be 1
6  "/chip.test", line 62: pin 64 should be 1
7  "/chip.test", line 62: pin 2 should be 1
8  "/chip.test", line 62: pin 3 should be 0
9  "/chip.test", line 62: pin 6 should be 0
10 "/chip.test", line 62: pin 10 should be 0
11 "/chip.test", line 62: pin 15 should be 0
12 "/chip.test", line 62: pin 16 should be 0
13 "/chip.test", line 67: pin 61 should be 1
14 "/chip.test", line 70: pin 16 should be 0
```

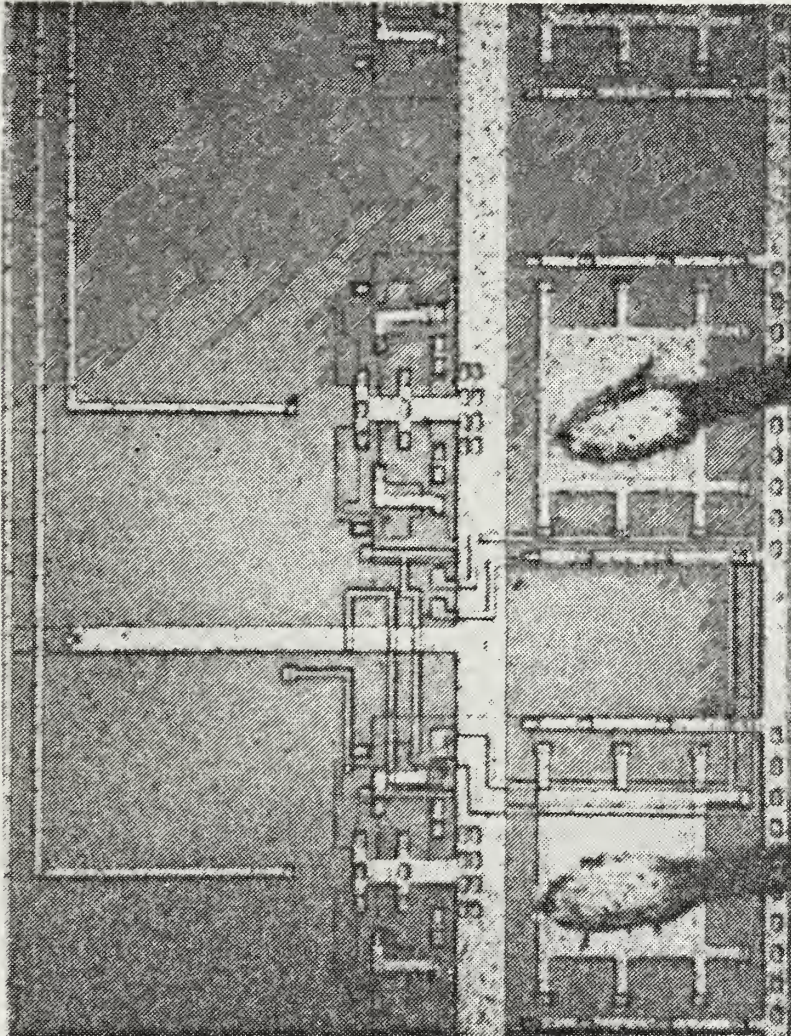
Figure(7.4) Error cues

C. RESULTS

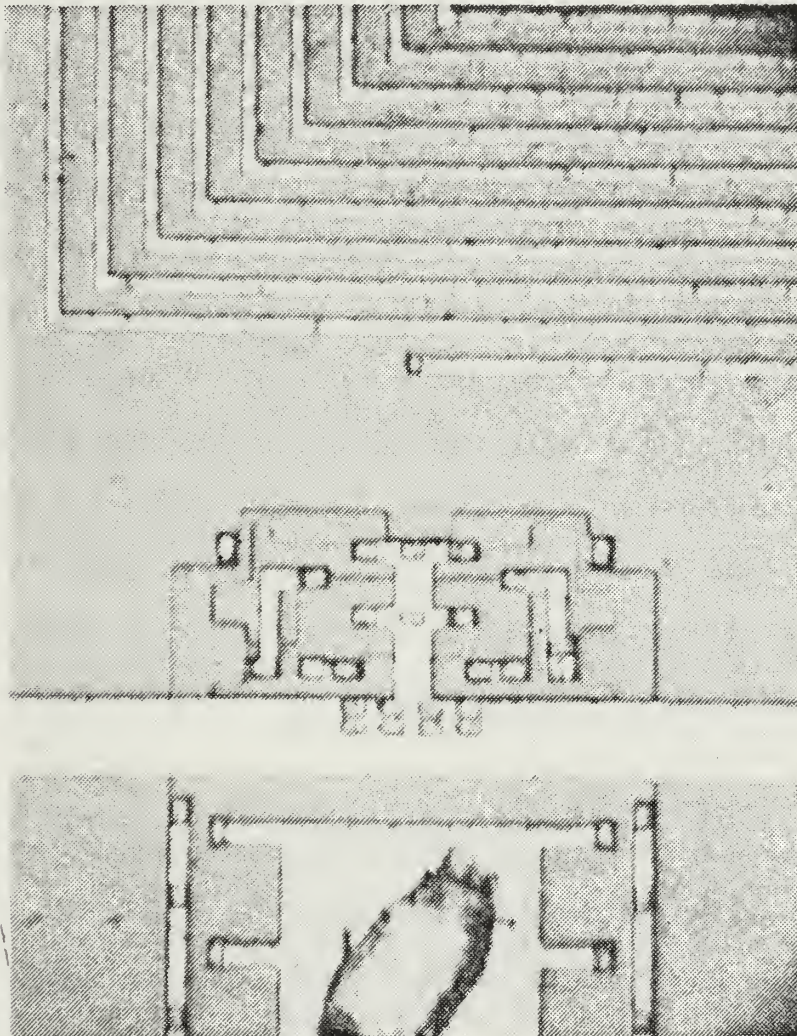
During step one of the test plan, it was discovered that on all eight chips four of the output pins remained in the "high" state and twelve remained in the "low" state continuously regardless of the input vectors. This indication, along with the fact that all eight chips drew approximately two-hundred milliamps over the

normal amount of current for a chip of this size, suggested serious problems. Microscopic inspection of the chip proved this to be the case. A large amount of the polysilicon "runs" were missing or shifted. The photographs shown in Figures(7.5 and 7.6) point out just a few of the many fabrication errors that rendered these chips totally useless. Figure(7.5) shows two output pads. Located between these two pads is a pattern formed of polysilicon. This "poly" pattern should be directly on top of both of the pads. If it were, the two nearby metal wire "runs" would connect properly to the pads and the output pads would operate correctly. It appears that VDD is shorted to GND through this misplaced "poly" pattern. Figure(7.6) also shows an output pad that is completely missing the polysilicon layer.

With gross fabrication errors of this nature, these chips had no chance of producing any viable results. Since the design has passed the DRC and the simulation, there is a very good chance that, if properly fabricated, a "good" chip will result.



Figure(7.5) Fabrication errors



Figure(7.6) Fabrication errors

VIII. CONCLUSION

A. SUMMARY

The objective of this thesis was to describe the use of VLSI CAD tools available at NPS and to provide a non-trivial example of design and implementation of a VLSI circuit using these tools. The tutorials in Chapters 3 through 5 have provided the necessary background to familiarize the designer with the available CAD tools. Suggestions were made to lessen the difficulties and examples were provided to highlight the proper usage of the tools.

The design and implementation of the actual thesis project (16-Bit Adder) was covered in Chapter 6. This Chapter provided a thorough example of VLSI design techniques and CAD tool implementation. The results of testing the fabricated chip was covered in Chapter 7. This Chapter indicated that the project was unsuccessfully manufactured so that evaluation of the design was impossible. However, since the submitted design passed all of the NPS VLSI validation tests (**drc**, and **esim**), there is a high probability that the design is sound.

B. RECOMMENDATIONS

The following recommendations should be taken into consideration:

1. Re-submit the CIF file for the adder for fabrication and test the returned chip for design accuracy. (Note: This has been initiated with fabrication beginning on October 6, 1983.)
2. Design a multiplier chip to be used in conjunction with the adder for implementation in a digital filter circuit.

3. Initiate a VLSI design course based on the contents of this thesis and Reference 1 in which students can combine efforts (or work individually) to generate CIF files of validated design circuits for fabrication and testing.

4. Continue software development in the area of VLSI CAD. Although the basis of the CAD tools has been established, several programs have not been investigated. The MIT software provides many such programs with the timing simulator (**rnl**) taking priority. Additionally the Berkeley software has been totally ignored (with the exception of (**esim**) due to the unavailability of the necessary graphics terminal.

APPENDIX A

INTRODUCTION TO THE VAX-11/780 AND UNIX

The Very Large Scale Integration (VLSI) Computer Aided Design (CAD) tools at the Naval Postgraduate School (NPS) have been assembled from a collection of software developed by various universities, including Stanford and Massachusetts Institute of Technology (MIT). Since this software was developed and tested for use on the Digital Equipment Corporation (DEC) VAX-11/780 computer, this computer was chosen for the NPS VLSI design implementation. This system uses the Berkeley UNIX 4.1 operating system.

A. THE COMPUTER

The VAX used for VLSI design is operated and maintained by the Computer Science (CS) Department (located in Spanagel Hall (SP) room SP-500) but has memory space and computer time allocated for the Electrical Engineering (EE) Department. The present system contains 2 megabytes of physical memory with plans to increase this in the near future. The VAX-11/780 is a general-purpose computer lying between minis and maxis in performance. Its power lies in its usage of an increased virtual memory with a 32-bit address over that of its predecessor (the PDP-11), hence its name - Virtual Address Extension (VAX). It has a virtual address space of about 4.3 gigabytes. VAX systems are highly reliable. Built-in protection mechanisms in both hardware and software ensure data integrity and system availability.

In order to be able to use the computer to design a VLSI circuit or system, a few basic concepts and procedures must be understood. The following sections provide background for the VLSI designer to work effectively with the computer system.

1. *Obtaining An Account*

To obtain an account on this computer, inquire in the CS Office (SP-515). Once a need has been established, an account and a password will be assigned. Additionally, the combination for the cipher lock of the terminal room should be obtained. You are now ready to locate a terminal and familiarize yourself with the system.

2. *Terminal Room*

The public access terminals of the VAX computers are located in room SP-511. The terminals used for VLSI design face the windows in the north wall (to the left when entering). There are five ADM36 terminals and one GIGI terminal available for public use. The GIGI terminal is capable of color graphic displays as well as black and white coding. The printer for the computer is located in room SP-500 (the computer room) and can be accessed through the door in the south wall of SP-511.

Use of the terminal room is controlled by the CS Department. The room is usually open from 0800 to 1630 on normal work days. At all other hours, the door is locked with a cipher lock.

The following rules apply to the terminal and computer rooms:

- * Ensure that the cipher locks are locked during non-working hours.
- * Prior to leaving; logout, turn the terminal off and clean up the area.
- * After working hours, secure the area by turning off the printers and lights. (Provided no other users are using the area.)
- * NO SMOKING in the terminal or computer rooms.
- * Place excess computer paper neatly in the available boxes for recycling.

3. *Login/Logout Procedures*

The master ON/OFF switch for the ADM36 terminal is located on the lower, right, back of the video monitor. (If the GIGI terminal is being used, there are two ON/OFF switches. The switch for the terminal/keyboard is located on the back, left of the keyboard and the switch for the monitor is located on the

upper, right, front of that unit.)

To login, turn on a terminal. After a short warm-up time and a cursor has appeared on the display, hit the RETURN key (<CR>). You should see

login:

If this prompt does not appear or if a strange display occurs try one or all of the following:

- Type logout then hit RETURN.
- Turn the terminal OFF then back ON. Hit RETURN.
- Seek help from one of the technicians.

If the login prompt appears, type in your account code (usually your last name) and hit RETURN. The screen should now display

login:
password:

Type in your account password (usually your last name) and hit RETURN. In order to protect your access, this entry is not displayed on the screen. If you make a mistake, the following will be returned

Login incorrect
login:

After a correct login has been completed, the system will display several lines of information for the user. The next prompt for the user will be

TERM = (vt100)

This is a request for the terminal type that is being used. All terminals in the terminal room have a vt100 display format, so simply hit RETURN. (If you are

using a terminal from a remote location via the dial-up system, type in the type of terminal or type in *tty* then hit RETURN.) At this point, the system displays a list of the current users and then stops with the system prompt

%

The percent sign (%) is the UNIX prompt which indicates that the computer is ready for a system command. Some of the more useful commands are presented in the section *Tutorial of UNIX commands*.

When you have finished using the computer, sign off by typing

logout <CR>

and the system will display the login prompt again. To secure the terminal, simply turn off the power switch(es). Then clean up the immediate area.

4. *Editing with "vi"*

The most popular text editor for the UNIX system is "vi." To familiarize yourself with this editor, login then type

vi.tutorial <CR>

The display will give you prompts to complete the self-paced tutorial on the "vi" editor. This tutorial will take a few hours, but will be worth the time in the long run. Once you are familiar with this editor, you are ready to learn about the other system commands.

5. *Tutorial of UNIX Commands*

After logging into the system, the UNIX prompt (%) indicates that the system is ready for a command. In this section, a selection of valid commands will be presented in order to familiarize you with the computer's responses. (This is

not intended to be a complete list of valid commands but is an introduction to the more commonly used ones. For a more complete tutorial, see Reference 2.)

Prior to trying the commands, a few general comments are necessary. In issuing any command, if the system "locks up" or if the display appears unusual in any way, try one (or all) of the following:

- * Press the BREAK key then hit RETURN.
- * Simultaneously press the CTRL and C keys (<CTRL>C).
- * Press the SET-UP key then the 0 (zero) key then hit RETURN.

To correct an input error, press the CTRL and H keys simultaneously (<CTRL>H). This will cause the cursor to back up over the previous character typed. (The character may not be erased but has been eliminated from the computer memory.) To eliminate a complete line from memory, press <CTRL>U. To stop a job that is in progress, press BREAK or <CTRL>C. It may be necessary to press RETURN to get the system prompt.

In the following tutorial, user inputs are in *italics* and must be followed by a RETURN. System commands are in **bold type**.

Prior to starting this tutorial, use **vi** to create the following files:

temp1:

```
111111
This is the first of two temporary
files to be used in this tutorial
111111
```

temp2:

```
222222
This is the second of two temporary
files to be used in this tutorial
222222
```

a. **Passwd**

To change your login **password** from an old password to a new password (This is advisable, since your initial password is usually the same as your login code word.), type

passwd

and the system will ask for the old password. If you type in the old password correctly, the system will request a new password. Finally, since none of the passwords are displayed on the screen, the system will verify that your entry is the password that you wanted by prompting you to type it again. The display should be

```
passwd
old password:
new password:
Retype new password:
```

and if the last two responses were identical, your new login password will be effective.

b. Mail

You can send or receive messages through the computer using the **mail** function. To send a letter to another user (or a reminder message to yourself), type

```
mail user-name
(Your message)
<CTRL>D (or .)
```

The CTRL and D keys when pressed simultaneously (or a period at the beginning of a line) will return you to the system prompt and automatically send your mail.

If mail is received, the computer indicates this fact with a message of

You have mail.

when you login, or a message of

New mail for (your-name) has arrived:

if you are logged on when the mail arrives. To read the mail, type

mail

A list of letters saved will be displayed and can be read using

p

The **p** command will print the first message in your mail box. To delete a letter, use the **d** command. To reply to a letter, use the **r** command. When you are finished using the **mail** function, you can exit to the system by typing

q

c. **Man**

To see the documentation of a UNIX command (to determine its correct usage), type

man (command)

and that command's description from section 1 of the *UNIX Programmer's Manual* will appear on the screen. For example, try

man mail

The manual page for the **mail** command will appear on the screen. Hitting the SPACE bar (<SP>) advances the output to the next block of lines until the end of

the manual is reached. To exit from **man** prior to reaching the end of the selected manual page, hit *q*.

d. **More**

The best method to display the contents of a file for the sole purpose of reading it is to use the **more** command. This function produces a display in the same format as **man**, but is used for files other than manual pages. Try typing

```
more vi.tutorial
```

If the file "vi.tutorial" is in your directory (it should be if you attempted the **vi** tutorial of the previous section), then the first block of lines for that file will appear on the screen. You will also observe a white block in the lower, left corner of the display. This indicates the percentage of the file that has been examined. As with the **man** command, pressing <SP> will advance the display and the function can be ended with *q*.

e. **Who**

Now try typing

```
who
```

You should see a list of the users who are currently logged on to the UNIX system. This is a good time to look at one of the system's special functions, the **pipeline** function (**|**). As an example of pipelining, type

```
who | sort
```

The system response is a sorted (alphabetically) version of the list of users. The **pipeline** command sends the output of the left argument into the right

argument.

If you type

who am i

the system will respond with your login name, terminal and time of login.

f. **Tty**

To see your terminal designation, type

tty

The computer will respond with your terminal name.

g. **Pwd**

The **pwd** command requests the name of your present working directory. This is the directory in which you are working. Type

pwd

and the computer will respond with

/work/your-name

This is your login directory which is under the directory "work" which is in the root directory "/" (There is more on directories and their hierarchy in the UNIX section of this appendix.)

h. **Cd**

To change your present working directory, you must use the **cd** command. Try typing

cd /work

You have now moved to the "work" directory which can be verified by typing

pwd

The response of the computer should be

/work

which tells you that you have moved into the "work" directory.

Now type

cd
pwd

The **cd** command, when used without an argument, sends you to the default directory which is your login directory.

Another way to change directories is by typing

cd .. (*two periods after the cd*)

which moves you up one directory in the hierarchy. If this command was typed while in your login directory, it would move you to the "work" directory. Try it, then do a *cd* to get back into your login directory.

i. **Date**

If you wish to see the current time and date, type

date

and the current time and date will be displayed on the screen.

j. **Ls**

The **ls** command causes a list of all files and directories in your working directory to be displayed on the screen. The general form is

```
ls
```

When this command is typed, the files in your present working directory will appear on the screen in alphabetical order.

Now try typing

```
ls -l
```

The **-l** is an option that controls the output of the **ls** command. The general form of an option is **-x** where **x** is the particular option(s) that you wish to activate. In this case the **l** option causes a long printout. Note that the display is now a long version of your working directory's files. This output provides the total number of 512-character blocks, the permission mode for each file or directory, the number of links, owner, size in bytes (or characters), and the date and time of the last modification for each file.

Another use of **ls** is for listing files in another directory. Try typing

```
ls /work
```

The result is a display of the files and directories in the "work" directory. If you examine this output, you will see your user name along with all of the other user names for those who have accounts on the UNIX system.

k. **Chmod**

A command that may be of some use later is **chmod**. This allows you to **change** the protection permission **modes** on any file or directory that you

own. The permission modes (as displayed with the `ls -l` command) indicate who can access or modify a given file. These modes can normally only be changed by the owner of the file. When a file is created (as with `vi`), the default permission modes are

```
-rw-r--r--
```

which means that the file is a plain file (the first `-`) and the owner has the permission to "read from" or "write to" the file (`rw`). Additionally, any users in your group (as determined by the login type) have permission to "read from" the file (the second `r`) and all other users also have the "read only" permission (the third `r`).

For example, type

```
ls -l temp1
```

and observe the permission modes. Now type

```
chmod +x temp1
```

To see the effect, type

```
ls -l temp1
```

You will observe that the permission modes have been changed to

```
-rwxr-xr-x
```

which gives all users of the system permission to execute (`x`) your file. Since this is not an executable file, change the permission modes back again with


```
chmod -x temp1
```

To see the effect, type

```
ls -l temp1
```

Now ensure that the modes have indeed been changed. A complete list of the **chmod** options can be obtained from Reference 2 or the *UNIX Programmer's Manual* [Ref. 3].

1. **Cp**

To **copy** a file from one directory into another or simply to make another copy of a file in the same directory, use the **cp** command. As an example, type

```
cp temp1 temp  
more temp
```

You will observe that "temp" is an exact replica of temp1.

Now try typing

```
cp temp1 /work/temp
```

The computer returns an error message of

```
cp: cannot create /work/temp
```

Since you don't own the directory "work" and don't have permission to "write to" that directory, you can't create the file "temp." If you owned another directory, this would be the method to copy a file into it. If the name of the file ("temp" in this case) is omitted from the command, the default would be to keep the same name in the copy.

m. **Mv**

The **mv** command is identical to the **cp** command with the exception that it **m**oves (vice copies) the named file into the named destination. Try typing

```
mv temp new
```

To see the effects, type

```
ls
```

You now have a file labeled "new" and the "temp" file has been eliminated. You can also move a file into another directory in the same manner as it can be copied.

n. **Rm**

To **r**emove a file from a directory, use the **rm** command. Type

```
rm new
```

To see the effect, type

```
ls
```

You have now removed the file "new" from your directory. You can only remove files for which you have "write" permission.

o. **Lpr**

The **lpr** command sends a named file to the line **p**rinter queue. Try typing

```
lpr temp2
```


If you now go into the computer room, the nearest printer should have printed the file "temp2."

It will probably be necessary to advance the printer so that your output can be torn off. To do this, put the printer in standby (off line) by pressing the ON/OFF LINE button. Then, advance the output by pressing the TOP OF FORM button. Now, put the printer back on line by pressing the ON/OFF LINE button.

If your file is not printed, make sure that the printer is on line then type (at your terminal)

```
clearprinter
```

This procedure should enable your output. If this does not work, call for help.

p. **Cat**

The **cat** function (short for **concatenate**) successively displays the contents of one or more files. The resultant output can be directed into another file or displayed on the terminal. To display the file "temp1," type

```
cat temp1
```

and the contents of that file will be displayed on the screen. Note that for long files, the NO SCROLL key can be used to stop the display from scrolling.

To use the same command to combine two files into a third file, type

```
cat temp1 temp2 > temp3
```

The result of the concatenation has now moved directly into a file "temp3." This file can now be inspected using


```
cat temp3
```

The result will be a screen display of the combination of "temp1" and "temp2." Note that the **directive (>)** causes the result of the left argument to be placed into the right argument.

q. **Mkdir**

If you have the need to **make** a **directory** under your login one, use the **mkdir** command. Try typing

```
mkdir report
```

To see the effect, type

```
ls -l
```

You have now created a directory "report" under your login directory.

r. **At**

To execute a file at a later time (even after you have logged off), use the **at** command. As an example of the use of this command, first create an executable file (using **vi**) called "delay."

delay:

```
echo "The file delay has been executed!";
```

If you now look at the permission modes for this file (using **ls -l delay**), you will see that it is not authorized for execution. Therefore, do a mode change

```
chmod +x delay
```

Now, type in the **at** command with a time 5 or 10 minutes from the present time using

at (time-to-execute) delay

You can now continue with the tutorial and the specified message will print out on the terminal at the designated time. (**echo** is a *c-shell* command and will be covered later.)

s. **Ps**

The **process status** command (**ps**) is used to provide status information for the processes that are currently active. In order to demonstrate this function, we will start a process "in the background" using the "and" sign (&). A good process to start is **sort** since it will last a long time. If you type

```
sort -r /usr/dict/words -o word.sort &
```

the computer will respond with a process number for this sort. Now type

```
ps
```

The display will indicate what processes are in progress and will give the corresponding process number. To stop the **sort** routine, continue on to the **kill** section. (Note that the **sort** function is explained further in the *UNIX Programmer's Manual* ; however, an understanding of it is not necessary for this tutorial.)

t. **Kill**

To stop a process that is in progress "in the background," use the **kill** command. In order to stop the word sort initiated in the previous paragraph, type

```
kill (process-number)
```


where "process-number" is the number obtained from the previous **ps** command. If you now type

ps

the computer output should not include the **sort** routine.

u. History

The **history** command is very useful if you wish to repeat a previously executed command. It is a *c-shell* command that provides a list of the commands that were executed since login. Type

history

and note that a number is assigned to each command that was executed. To execute any of these commands again, simply type

!(number)

where *number* is the number of the command you wish to execute.

Another way to re-execute a command is

!x

where *x* is the first letter(s) of the desired command. For example, type

cat temp2

and the "temp2" file will print out. Now type

!c

and the computer will respond with the full command followed with the print out of "temp2." A note on the method of issuing a command -- The computer searches your previous commands from latest issued to first; therefore, if you specify a command with only one letter, the last command starting with that letter is executed.

This concludes the tutorial. Although not all commands were addressed, you should have enough experience to use the UNIX system for VLSI design. Try experimenting with various commands to see the result. Experience and trial-and-error are the most effective ingredients to learning the UNIX system.

B. THE UNIX OPERATING SYSTEM

The UNIX operating system is a very complex, but flexible, system which gives the experienced user a powerful tool toward writing successful programs. Now that you have enough experience to use the computer, a closer look into the operating system will probably round out your knowledge and help to make your use of the computer for VLSI design a little easier.

1. *Hierarchy*

The UNIX system uses a hierarchical approach to file management. The "root" directory (/) is the starting point for this arrangement with all other directories and files stemming from it. Under the "root" directory is the "work" directory which contains all of the "login" directories for the users of the system. It is in this "login" directory that you will start your own hierarchy of files and directories. Each file (or directory) that you form will stem from your "login" directory. There is no set format for this hierarchy, so it is left up to the user to form a structure that will best benefit him.

2. *Manuals*

The manuals for the Unix operating system are located on the tables in the terminal room (SP-511). Although these manuals are quite extensive, they are well written and provide all the information that you will need to operate the system. The manuals are grouped together on one rack and are separated with heavy dividers. They are labeled *UNIX PROGRAMMER'S MANUAL* and consist of the main manual (Volume 1) and three volumes of supporting data (Volumes 2A, 2B, and 2C).

Volume 1 contains all of the valid UNIX commands and is the most commonly used. This volume is divided into eight sections:

1. Commands
2. System Calls
3. Subroutines
4. Devices & Special Files
5. File Formats & Conventions
6. Games
7. Macro Packages & Language Conventions
8. Maintenance

Of these eight sections, the first three will be of the most help to the average user. In these sections are the correct usage of the general system commands and routines. Although it appears that this volume is too extensive to be of much use, the *Permuted Index* starting on page *xxiii* makes it easy to locate any command that is of interest.

For example, try to locate the manual page for the **mail** command. Looking in the *Permuted Index* under "mail" (note the alphabetical order) you will see an entry

mail: send and receive mail.....mail(1)

This entry tells you that the **mail** command is located in Section 1 (i.e., it is a

command). Now, if you look in that section of Volume 1, you will find the manual page for this command.

Volume 2A is the initial supplement to the Programmer's Manual. It provides information to help the beginner get started using the UNIX system and the "C" language.

Volume 2B is an extension of 2A covering special features of the operating system.

Volume 2C is the second extension of 2A. It covers the editing routines as well as programming in the *c-shell*.

3. *C-Shell*

The *c-shell* is a command language interpreter used by the UNIX system. It is described fully in Volume 2 (A and C) of the Programmer's Manual, so I won't spend a lot of time on it. However, if you recall from the section *Tutorial of UNIX Commands* under the tutorial on *at*, you initiated a *c-shell* command of *echo*. This was actually programming using *c-shell*. Although it may not seem apparent now, this type of programming can be an invaluable aid to you. For example, if you have a series of commands that you wish to execute in order (especially if you need to repeat the series often), you can write a program in *c-shell* containing those commands and then you will need to execute that one program only. As an example, this could be a file "combo"

```
who > store
sort < store > out
cat out
```

If this file is executed, the result would be a sorted version of the system users who are logged on. This is a trivial example but should serve as a introduction to programming in *c-shell*.

4. *Introduction To Programming In "C"*

The UNIX operating system was designed to accept programs in a programming language called "C". Because of this fact, the data files necessary to complete a VLSI design use the "C" format. Although it is not necessary for the designer to be an expert "C" programmer, a basic understanding of this language will be helpful.

The general format for a "C" program is:

```
file.c:      /* This is a basic C program */
              main()
              {
                  printf("This is test print out 0");
              }
```

In the above example, you can see that comments are set off by "*/* */*." The declaration "*main()*" indicates that the main function of this program is labeled "main" and has no arguments. The function definition begins with "{" and ends with "}". Within the main function is another function "printf" which has as its argument the sentence within the quotation marks. "Printf" is a system subroutine (section 3 of the *UNIX Programmer's Manual*) which causes the argument to be printed on the standard output (the terminal). Each program statement must end with a semicolon (;).

"C" is a "free-form language that doesn't care what style or format you use, as long as it is syntactically correct." [Ref. 4, p. 11] However, indentation can and should be used to make the program easier to read. Most of the statements are written in lower-case letters with the exception of symbolic names and constants.

To compile the example "C" program, type

```
cc file.c
```

The result is an executable file called "file." (**cc** is the "C" Compiler and is documented under section 1 of the Programmer's Manual.) To execute this file, type

```
file
```

and the specified statement will appear on the screen.

This has been a very brief introduction to "C" but should provide enough basics for the user to continue with learning VLSI design. If more information is desired, consult the *UNIX Programmer's Manual* [Ref. 3] or Reference 4.

APPENDIX B
MANUAL PAGES FOR VLSI CAD TOOLS

NAME

cif — convert a *cif* file to *cifout* format

SYNOPSIS

cif *file.cif* [-o *file.co*] [-qbcdgimnpq] [-y] [-z] [-h]

DESCRIPTION

Cif converts a *cif* file to *cifout* format. Cif files, which are the Caltech Intermediate Format, are described in the **Mead and Conway** text.

The **-o** option specifies the name of the output file. If not given, the output file has the same name as the input file and the extension ".co".

The **-q** option says that the given layer names are not valid. For example, in the standard nMOS you would use -qbnq since buried contacts and two level metal and poly are not allowed.

The **-y** option says that the Y layer is valid and should be replaced by the Z layer. This allows preprocessors to display the outline of a box with no internal stipples.

The **-z** option suppresses the printing of warning messages about zero area rectangles.

The **-h** option causes CIF to list node numbers and lower left corners of all rectangles with nonzero node numbers to the error device.

FILES**SEE ALSO**

cifout(cad5)

BUGS

NAME

cifar — save cif files in archive format suitable for use by **cifload**

SYNOPSIS

cifar [options] file.lib file.cif

DESCRIPTION

Cifar prepares an archive file of CIF cells suitable for use with **cifload**. The standard use is to have the input **CIF** file split up into archive units each containing one cell. This is useful for libraries which do not have the external and entry point records present. **Cifar** can also be used to archive **CLL** produced files, which already contain the information necessary for **CIFLOAD**.

-a[ar options]

Letters following are options used to control archive program *ar*. This switch is **required**.

-u Specifies that the CIF file given is not to be split up into individual cells, but that it is to be entered into archive file as a unit. The external and entry records must already exist in the file if this option is used.

A file to be processed by **cifar** must have all calls, DS and DF commands as the first character of the line. The required linkage is specified in comments occurring before the DS. Each comment starts in column 1 of the line. The command (**ext** <number>); specifies that the following cell requires the cell named <number>. There may be several of these before a cell. The command (**ent** <number>); specifies that the next subfile contains the cell named <number>. Since several cells can be contained in a single archive subfile, there may be more than one.

FILES

atmpXXXXXX, ctmpXXXXXX	temporary files
[0-9]*.ctmp	temporary files containing CIF cells
/vlsi/lib/local/splitfile	splits CIF file into separate files
/vlsi/lib/local/cifar	
ar	system archiver

SEE ALSO

cif(cad1), cl12(cad1), cifload(cad1), ar(1)

DIAGNOSTICS

Diagnostics may come from **cifar**, **splitfile** or **ar**.

AUTHOR

Wayne H. Wolf, Esq.

BUGS

Places temporary files on your directory. All files on your directory with names *.ctmp are deleted.

NAME

cifload – concatenates cif files and needed library cells from archive files

SYNOPSIS

cifload [options] ...file.lib... ...file.cif...

DESCRIPTION

Cifload searches libraries for cells needed by **CIF** files. The input **CIF** files contain records at their head declaring which library cells they need; each library, maintained in archive format, contains a set of files with declarations of what cells they contain. **Cifload** does not guarantee that it will satisfy all externals. This is done to alleviate the problem of deciding what externals are satisfied by other **CIF** files rather than library files. The linked set of cells appear on the standard output.

Options:

- States that the standard input also contains a **CIF** file. This file will be made the last file to appear on the standard output; therefore it should contain the final cell call and the end statement.

FILES

/vlsi/lib/local/cifload

SEE ALSO

cif(cad1), cll2(cad1), cifar(cad1), ar(1)

DIAGNOSTICS

Complains if there are no **CIF** files input. May also blow up if there are a large number of external references in a single file.

AUTHOR

Wayne H. Wolf, Esq.

BUGS

Can handle only a limited number of external references or entry points from a single file. The seeking of the next archive header is done in a slow manner because the nature of the archive file is not well documented.

NAME

file.co output of CIF translator

DESCRIPTION

A *cifout* file is produced by the CIF translation program. The file represents an integrated circuit as a collection of rectangles with layer information for each rectangle. The rectangle information is written in a binary format. There is also some control information embedded within the file. This information is always at the head of the file and is written in ASCII.

All control lines start with "#" as a key, and all control lines must be collected at the beginning of the file. The last control line must be #end. The maximum length of a control line is currently 80 characters. Immediately following the "#" is a keyword. The keywords are program specific and consequently subject to future improvements. Currently used keys are:

<space>	for comments
bounds	minimum_x minimum_y delta_x delta_y
file	input file name
noprint	tells rplot to not print the comment lines
document	tells rplot to print on 8.5 x 11 inch paper
report	tells rplot to print on 8.5 x 11 inch paper with margin for hole punch
scale	scale to be used in plot, in lambda/inch
noscale	data is not to be scaled before plotting

Following the control lines, if any, are data lines. There is one data line per rectangle. The records are written in binary form, and are to be accessed only through standard procedures. The access procedures return floating point numbers, but the file is currently accurate only to within .5 lambda. The layer is returned as a character. Current layers are:

C Contacts
 D Diffusion
 G Glass
 I Implant
 M Metal
 P Poly

Z Unknown The node number is character. If the node number = 99 then the rectangle is at a 45 degree angle (rotated clockwise about the lower left corner).

FILES

/vlsi/lib/local/cifout	
/vlsi/stanford/src/cif/cifout-data.h	definitions for cifout i/o
/vlsi/stanford/src/cif/cifout-io.h	standard i/o routines
/vlsi/stanford/src/cifplot/scale.h	scaling package for plotting programs
/vlsi/stanford/src/cifplot/scale-factor.h	device-dependent parameters for

scaling

SEE ALSO

`cif(cad1)`, `window(cad1)`, `rrplot(cad1)`

NAME

cll - process cll, cif, and sco files, plotting the output

SYNOPSIS

cll [options] ... file ...

DESCRIPTION

cll is the VLSI project's CLL language processor. It accepts several types of arguments:

- 1) Arguments whose names end with ".cll" are taken to be source files in CLL.
- 2) Arguments whose names end with ".cif" are taken to be source files in CIF.
- 3) Arguments whose names end with ".co" are taken to be *cifout(V)* files.
- 4) Arguments whose names end with ".sco" are taken to be sorted *cifout(V)* files.
- 5) Arguments that start with "-" are taken to be switches.

The basic operation of **cll** is to process the cell library externals and the ".cll" files in order, creating a CIF file. This file is then processed together with the CIF for the cell library and the ".cif" files to create a *cifout(V)* file. It is possible to start with a single ".co" file, instead of using ".cll" or ".cif" files. Finally, any ".sco" files are overlaid on top, and the final *cifout* file is then plotted.

The processing can be modified by the following switches:

- lx** Include cif library **libx.cif**.
- b** Specified area is blocked out (not plotted).
- c#** Produce a ".cif" file without a final call or end statement. If # is present, use cif numbers #, #+1, etc. Such a file is suitable for reprocessing by **cll**.
- C** Process all ".cll" and ".cif" files into a single ".cif" file with the specified cell libraries and a final call and end. This file will be suitable for fabrication.
- d** Plot is formatted to fit in document style (8.5 X 11 in.).
- P** Plot the output on the Versatec plotter.
- r** Plot is formatted in report style (room for binding).
- gx.y** Plot a grid whose x interval is *x* lambda and whose y interval is *y* lambda. (Default interval is 5 lambda).
- i#** Use a scale factor of # lambdas per inch.
- nl** Causes the named layer, *l*, to be omitted from the plot. The layer, *l*, can be one of more of: **c, d, g, i, m, p**.
- x#1.#2**
Set the minimum x to be plotted as #1 lambda and the maximum to #2 lambda. Either #1 or #2 can be omitted, in which case the actual minimum or maximum will be used.
- y#1.#2**
Set the minimum y to be plotted as #1 lambda and the maximum to #2 lambda. Either #1 or #2 can be omitted, in which case the actual minimum or maximum will be used.
- s#1.#2**

Divides your chip into #2 strips and plots the #1'th strip.

- S# Divides your chip into # strips, and plots all of those strips.
- F Create final version of project. The -F switch sets the -C switch, which will cause *cll* to create a complete *cif* file. A special feature of the -F switch is that the output file name is *./final.cif* and all optimization is switched off.
- T Plot the output on your terminal, using tplot. You have to be using a GIGI terminal.
- X Just do the pre-processing pass creating a *./xccl* file.
- B Use backup *cif* expander. (This is for testing, don't try it yourself.)
- A Use alternate copy of *cil2* language processor. (This is for testing, don't try it yourself.)
- D Trace operation of *cil*.
- Z Use alternate CIF loader (that is, *cat*).

FILES

/vlsi/lib/local/libx.cif	cif for the cell libraries
/vlsi/tmp/cil?????	temporary
/lib/cpp	preprocessor
/vlsi/lib/local/cil	CLL source program
/vlsi/lib/local/cil2	CLL language processor
/vlsi/lib/local/acil2	Alternate CLL language processor
/vlsi/lib/local/cifload	CIF linkage editor
/vlsi/lib/local/cif	CIF language processor
/vlsi/lib/local/acif	Alternate CIF processor
/vlsi/lib/local/rsort	sorts cifout files
/vlsi/lib/local/merge	merges cifout files
/vlsi/lib/local/window	windows cifout files
/vlsi/lib/local/rplot	plots on the Versatec
/vlsi/lib/local/tplot	plots on GIGI terminals
/vlsi/lib/include	include files
/vlsi/stanford/src/cil/pathnames.h	actual names of files used

SEE ALSO

Tim Saxe, *CLL - A Chip Layout Language*.
cif(cad1), *window(cad1)*, *tplot(cad1)*, *rplot(cad1)*, *cil2(cad1)*

DIAGNOSTICS

The diagnostics from CLL and CIF are supposed to be self-explanatory. However, *syntax error* often occurs for odd reasons. The normal solution is to correct all of the errors that you can easily locate and try again. Note that a plot will not be generated until the CLL and CIF processors are completely happy.

BUGS

No geometrical or circuit error detection or correction. What you say is what you get.

NAME

convert — converts a binary file to ASCII

SYNOPSIS

convert < *file*

DESCRIPTION

Convert takes a binary cifout file from standard input and converts it to a readable ASCII format sent to standard output.

FILES

/vlsi/lib/local/convert

SEE ALSO

cifout(cad5), unconvert(cad1)

BUGS

NAME

drc - design-rule-check a circuit

SYNOPSIS

drc file [**shift**]

DESCRIPTION

Drc does a design rule check of the input file. The file must be a sorted cifout file (with a .sco extension). This is done with *cil*. The output goes to *file.drc*.

Drc will check for Mead & Conway design rule violations with one general exception. Electrically connected areas will not generate separation errors, even if they are on different layers. In other words, *drc* will *not* enforce the 1 lambda separation between poly and diff if they are electrically connected. This means that a 2 lambda wide diffusion wire can run along a polysilicon wire, which is dangerous. A mis-alignment, during fabrication, of the polysilicon over the diffusion will increase the diffusion resistance, which can be bad if the overlap is very long.

The *shift* option will simulate a possible fabrication mis-alignment and do a more conservative check. It does this by expanding the poly layer by 1 lambda and then removing the expanded layer from the diffusion layer before doing diffusion minimum width checks. The default check, *no shift* option, is consistent with the rule that diffusion only has to extend beyond transistors by 2 lambda, but the *shift* option allows a tighter check if you want it.

The output file format consists of a message followed by coordinates of design rule violations. For example, part of an output file might look like:

```
poly min width errors:
diff minwidth errors.
10, 20
11, 20
```

indicating there were no polysilicon minimum width errors, but there were diffusion minimum width errors. Note that one error can cause several coordinates to come out.

The messages are self explanatory, although there are several quirks. The *shift* option causes the most commonly misunderstood error: a "diff minwidth error" caused by a poly-diff spacing error. This typically occurs at butting contacts. This happens when an arbitrary one lambda polysilicon shift reduces the diffusion line width to less than two lambda.

Pullups with "wide" butting contacts can also cause confusing errors. This happens when the diffusion overlap at the butting contact is *wider* (more than four lambda) than really needed for the butting contact. This can produce transistor-poly surround errors, transistor-implant surround errors, and floating transistor drain errors. The solution is to only use as much poly-diff overlap as is necessary for the butting contact. Any extra overlap only adds unwanted capacitance anyway.

SEE ALSO

cil(cad1), *cifout*(cad5)

DIAGNOSTICS

If some part of the design rule checker fails, error messages will appear in the output (*.drc*) file.

BUGS

While *drc* is running it will produce many files of the form *file.xx*, where *xx* is any two letters. These files are deleted at the completion of the *drc*, but uncatchable signals (like *kill -9*) can stop the *drc* and leave the files around.

NAME

esim - event driven switch level simulator

SYNOPSIS

esim [*file1* [*file2* ...]]

DESCRIPTION

Esim is an event-driven switch level simulator for NMOS transistor circuits. *Esim* accepts commands from the user, executing each command before reading the next. Commands come in two flavors: those which manipulate the electrical network, and those to direct the simulation. Commands have the following simple syntax:

c arg1 arg2 ... argn <newline>

where 'c' is a single letter specifying the command to be performed and the *argi* are arguments to that command. The arguments are separated by spaces (or tabs) and the command is terminated by a <newline>.

To run *esim* type

esim *file1* *file2* ...

Esim will read and execute commands, first from *file1*, then *file2*, etc. If one of the file names is preceded by a '-', then that file becomes the new output file (the default output is stdout). For example,

esim *f.sim* -*f.out* *g.sim*

This would cause *esim* to read commands from *f.sim*, sending output to the default output. When *f.sim* was exhausted, *f.out* would become the new output file, and the commands in *g.sim* executed.

After all the files have been processed, and if the "q" command has not terminated the simulation run, *esim* will accept further commands from the user, prompting for each one like so:

sim>

The user can type individual commands or direct *esim* to another file using the "@" command:

sim> @ *patchfile.sim*

This command would cause *esim* to read commands from "patchfile.sim", returning to interactive input when the file was exhausted.

It is common to have an initial network file prepared by a node extractor with perhaps a patch file or two prepared by hand. After reading these files into the simulator, the user would then interactively direct *esim*. This could be accomplished as follows:

esim *file.sim* *patch.1* *patch.2*

After reading the files, *esim* would prompt for the first command. Or we could have typed:

% *esim* *file.sim*

sim> @ *patch.1*

sim> @ *patch.2*

Network Manipulation Commands

The electrical network to be simulated is made up of enhancement and depletion mode transistors interconnected by nodes. Components can be added to the network with the following commands:

e gate source drain

e gate source drain length width key xpos ypos area

Adds enhancement mode transistor to network with the specified gate, source, and drain nodes. The longer form includes size and

location information as provided by the node extractor -- when making patches the short form is usually used.

d gate source drain

d gate source drain length width key xpos ypos area

Like "e" except for depletion mode devices.

C node1 node2 cap

Increase the capacitance between *node1* and *node2* by *cap*. *Esim* ignores this unless either *node1* or *node2* is GND.

= node name1 name2 name3

Allows the user to specify synonyms for a given node. Used by the node extractor to relate user-provided node names to the node's internal name (usually just a number).

| comment...

Lines beginning with vertical bar are treated as comments and ignored -- useful for deleting pieces of network in node extractor output files.

i node

Input record -- output by node extractor and not used by *esim*.

Currently, there is no way to remove components from the network once they have been added. You must go back the input files and modify them (using the comment character) to exclude those components you wished removed. "N" records need not be included for new nodes the user wishes to patch into the network.

Simulator Commands

The user can specify which nodes are to have their values displayed after each simulation step:

w node1 -node2 node3 ...

Watch node1 and node3, stop watching node2. At the end of a simulation step, each watched node will displayed like so:

node1=0 node3=X ...

To remove a node from the watched list, preface its name with a '-' in a "w" command.

W label node1 node2 ... nodeN

Watch bit vector. The values of nodes node1, ..., nodeN will displayed as a bit vector:

label=01010020

where the first 0 is the value of node1, the first 1 the value of node2, etc. The number displayed to right is the value of the bit vector interpreted as a binary number; this is omitted if the vector contains an X value. There is no way to unwatch a bit vector.

Before each simulation step the user can force nodes to be either high (1) or low (0) inputs (an input's value cannot be changed by the simulator!):

h node1 node2 ..

Force each node on the argument list to be a high input. overrides previous input commands if necessary.

l node1 node2 ...

Like "h" except forces nodes to be a low input.

x node1 node2 ...

Removes nodes from whatever input list they happen to be on. The next simulation step will determine their correct value in the circuit. This is the default state of most nodes. Note that this does not force nodes to have an "X" value -- it simply removes them

from the input lists.

The current value of a node can be determined in several ways:

v

View. prints the values of all watched nodes and nodes on the high and low input lists.

? node1 node2 ...

Prints a synopsis of the named nodes including their current values and the state of all transistors that affect the value of these nodes. This is the most common way of wondering through the network in search of what went wrong...

! node1 node2 ...

For each node in the argument list, prints a list of transistors controlled by that node.

"?" and "!" allow the user to go both backwards and forwards through the network in search of that piece causing all the problems.

The simulator is invoked with the following commands:

s

Simulation step. Propagates new values for the inputs through the network, returns when the network has settled. If things don't settle, command will never terminate -- try the "w" and "D" commands to narrow down the problem.

c

Cycle once through the clock, as define by the **K** command.

I

Initialize. Circuits with state are often hard to initialize because the initial value of each node is X. To cure this problem, the **I** command finds each node whose value is charged-X and changes it to charged-0, then runs a simulation step. If one iterates the **I** command a couple times, this often leads to a stable initialized condition (indicated when an **I** command takes 0 events, i.e., the circuit is stable).

Try it -- if circuit does not become stable in 3 or 4 tries, this command is probably of no use.

Miscellaneous Commands

D

toggle debug switch. useful for debugging simulator and/or circuit. If debug switch is on, then during simulation step each time a watched node is encountered in some event, that fact is indicated to the user along with some event info. If a node keeps appearing in this prinout, chances are that its value is oscillating. Vice versa, if your circuit never settles (ie., it oscillates) , you can use the "D" and "w" commands to find the node(s) that are causing the problem.

> filename

write current state of each node into specified file. useful for make a break point in your simulation run. Only stores values so isn't really useful to "dump" a run for later use -- see "<" command.

< filename

read from specified file, reinitializing the value of each node as directed. Note that network must already exist and be identical to the network used to create the dump file with the ">" command.

These state saving commands are really provided so that complicated initializing sequences need only be simulated once.

L

invokes network processor that finds all subnets corresponding to simple logic gates and converts them into form that allows faster simulation. Often it does the right thing, leading to a 25% to 50% reduction in the time for a single step. [We know of one case where the transformation was not transparent, so caveat simulee...]

X...

call extension command -- provides for user extensions to simulator.

q

exit to system.

Local Extensions

V node vector

Define a vector of inputs for the node. The first element is initially set as the input for *node*. Set the next element of the vector as the input after a cycle.

R n

Run the simulator through n cycles. If n is not present make the run as long as the longest vector. All watch nodes are reported back as vectors.

N

Clear all previously defined input vectors.

K node1 vector1 node2 vector2 ... nodeN vectorN

Define the clock. Each cycle, nodes 1 through N must run through their respective vectors.

SEE ALSO

extr(cad1), sim(cad1)

BUGS

NAME

extract - circuit extractor for a CIF file

SYNOPSIS

extract file

DESCRIPTION

Extract is the first of a sequence of programs for setting up your design for functional simulation. The first step is to begin with a *.cif* file. This normally means executing the following *cil* command:

cil -C file.cil

Then execute *extract* and wait up to 2 hours!

extract file

The next step is to plot the extracted circuit using *node-plot*. The last step is to create a file which assigns names to important nodes; this will include *vdd* and *gnd*, and probably *phi1* and *phi2*. For example,

-----*file.sym*-----

```
178  vdd
84   gnd
17   phi1
414  phi2
15   s0
13   s1
11   s2
9    o0
7    o1
5    o2
```

-----end of sample-----

Then create the simulation file (*.sim*) using *sim*. The extracted circuit is now ready for a static test with *stat* to determine ratio errors and power-ground shorts and an actual simulation with *esim*.

FILES

```
/vlsi/lib/local/extr/extract
/vlsi/lib/local/extr/toced
/vlsi/lib/local/extr/expand
/vlsi/lib/local/extr/bsort
/vlsi/lib/local/extr/bbound
```

SEE ALSO

node-plot(cad1), *sim(cad1)*

BUGS

Generates several *.def* files which are not normally needed by the user.

NAME

`merge` — merge two or more cifout files

SYNOPSIS

`merge < file1 file2 file3 ... [-o outfile]`

DESCRIPTION

Merge does a merge of sorted cifout files or sorted and unsorted cifout files. The input must be binary data and the output is binary data to the standard output. If the `-o` option is used, the output is sent to the stated file. This file cannot have the same name as any of the input files.

FILES

`/vlsi/lib/local/merge`

SEE ALSO

`cifout(cad5)`, `rsort(cad1)`

BUGS

NAME

node-plot — generate plot of extracted circuit

SYNOPSIS

node-plot *file*

DESCRIPTION

Node-plot generates a plot of an extracted circuit. The plot is automatically broken into strips of 240 lambda width and has the node numbers that are associated with the various node locations. The first part of the file name is used for the input. For example, to plot an extracted circuit which has a *.rec* file labeled *test.rec*, enter **node-plot test** and the terminal will indicate the necessary response for the plot.

FILES

/vlsi/lib/local/extr/node-plot
/vlsi/lib/local/extr/bbound

SEE ALSO

extract(cad1)

BUGS

The scale factor cannot be adjusted by the user. The stipple pattern is different from the one used by the *cll* plot routine.

NAME

plagen — layout a PLA in CIF from an input-output specification

SYNOPSIS

plagen [options] input pla.cif

DESCRIPTION

Plagen is a program that converts an input-output specification for a PLA into a CIF representation of the PLA. The CIF representation uses the XEROX cell library, and thus has a high probability of working. Since many people require different inputs and outputs, *plagen* only generates the AND-OR plane with associated pullups.

The options for *plagen* are:

- o** Do not include pullups on the OR plane. This allows you to take outputs from the top of the OR plane.
- g#** Set frequency of grounds to #. The default is one ground per 32 product terms.
- i** The inputs are interleaved.
- c** The inputs are complemented.

To use *plagen* you must first create an input file that specifies the inputs and outputs of the PLA. The format of the input file is:

#_of_inputs, #_of_terms, #_of_outputs, symbol#, lambda

xxxxx yyy

xxxxx yyy

.

.

.

xxxxx yyy

where #_of_inputs is the number of inputs to the PLA

#_of_terms is the number of terms in the PLA

#_of_outputs is the number of outputs that the PLA has (If zero only the AND plane will be generated)

symbol# is the number that the CIF symbol will have. This is how different PLA cells can be distinguished. You must be careful when you select the symbol number. For instance, the XEROX library consumes CIF numbers 1 to 99 and other special cells developed at Stanford use the numbers 100 to 899. Since CLL generates CIF symbols with numbers 1000 and greater, I suggest that you use CIF symbol numbers in the range 900 to 999.

lambda should be the current value of lambda in micrometers.

The actual programming information is encoded in #_of_terms lines of input. Each term of the PLA has #_of_inputs characters that represent the input connection information (the x's), a single space, and #_of_outputs characters that represent the connections to the outputs. For the inputs connections, there are three possibilities:

- 1) this term does not depend on this input: use a "-"
- 2) this term is only true if the input is true: "1"
- 3) this term is only true if the input is false: "0"

For the outputs there are only two possibilities:

- 1) this output is affected by this term: "--"
- 2) this output is not affected by this term: "0"

For example, suppose we wish to create a 4 input, 3 output, 3 term PLA with defining equations:

$$z1 = A'BC' + BC$$

$$z2 = A'BC' + ABCD$$

$$z3 = ABCD + BC$$

If we choose symbol number 901 and lambda of 2.5, then the input file is:

4,3,3,901,2.5

010- --0

-11- -0-

1111 0--

The output of plagen is a CIF file, and a line of information about the PLA cell. *Plagen* sends to the terminal a line that is an external definition of the PLA for use with CLL. Of course, you may need to alter the name of the CLL symbol that corresponds to the PLA cell.

FILES

/vlsi/lib/local/plagen

SEE ALSO

c11(cad1), plague(cad1)

BUGS

Not much error checking on the input format.

NAME

plague - PLA g(enerator) u(sing) e(quations)

SYNOPSIS

plague <input | plagen >pla.cif

DESCRIPTION

This is a program for producing a file suitable for the program "plagen" from logic equations. The file fed to it should first contain a CIF number for the whole symbol written "CIF# x;" (defaults to 900 if left out), then a list of input pins of the form "in<puts>: a1 a2 ...;" where "puts" is optional, a list of output pins "out<puts>: o1 o2...;", and a series of equations of the form "outpin = inpin1&inpin2'&x + etc;". The pin names can be any combinations of letters, digits, ., and _ but must start with a letter. Logical inversion is expressed by a ' after the pin name. The logical AND operator is '&', and the logical OR '+'. The equations are assumed to be in sum of products form. The order of the names in the input and output lists determines where they are on the PLA. Spaces, tabs, and newlines are ignored in the equations, and they and the lists are terminated by semicolons.

The program does no minimization, but does ignore duplicate product terms. The output for the plagen program comes out on the standard output. A schematic version showing the pin names is put into pla.schem.

Example

Here is what the original input looks like:

```
CIF# 950;
outputs: S1 S2 S3 a4' inc;
in: random input signal RESET' s3 s2 s1;
inc = random&input;
a4' = s3 + signal + random&input;
S3 = s2&s1 + RESET; S1 = signal + s1'&s2'&s3'; S2 = random&s3' + s1;
```

Note that RESET' was used in true and inverted form.

This is what goes to plagen:

```
7,8,5,950,2.5
11---- 000--
----1-- 000-0
--1--- -00-0
----11 00-00
---0--- 00-00
----000 -0000
1---0-- 0-000
-----1 0-000
```

Here is pla.schem:

```
CIF number 950
AND plane
1-----1- random
1----- input
--1---- signal
----0--- RESET'
```



```
-1--00- s3
--1-0-- s2
--1-0-1 s1
OR plane
00-00-00 S1
000000-- S2
000--000 S3
--00000 a4'
-0000000 inc
```

A "1" in the AND plane means that this term is true only if the input is true, a "0" that this term is true only if the input is false, and a "-" is a don't care. In the OR plane, a "-" means the output is affected by this term, and a "0" that it is unaffected.

FILES

/vlsi/lib/local/plague

SEE ALSO

plagen(cad1)

BUGS

Limited to 40 input, 40 output and 150 product terms. Pin names are limited to 14 characters.

NAME

rplot — converts a scaled, sorted cifout file to raster format and plots it

SYNOPSIS

rplot [options]...file...

DESCRIPTION

Rplot takes rectangles as input and creates a raster file output. Input is on standard input and output is to the Versatec plotter. The data must be sorted by x-coordinate.

The processing can be modified by the following switches:

- b** Produce a banner at the beginning of the plot.
- d** Scale the output for 8.5 X 11 inch paper (document form).
- gx.y** Plot a grid whose x interval is *x* lambda and whose y interval is *y* lambda.
- i#** Use a scale factor of # lambdas per inch.
- nl** Cause the named layer, *l*, to be omitted from the plot. The layer can be one or more of: **c, d, g, i, m, p**.
- r** Produce a report document plot with room for binding.
- s** Send output to standard output.
- x#1.#2**
Set the minimum x to be plotted as #1 lambda and the maximum as #2 lambda.
- y#1.#2**
Set the minimum y to be plotted as #1 lambda and the maximum as #2 lambda.

If no indications of the area to plot are given, *rplot* will scale the plot to best fit the Versatec width (11 inches).

FILES

/vlsi/lib/local/rplot

SEE ALSO

cifout(cad5), window(cad1), rsort(cad1)

BUGS

Rplot does not use the standard queue for the Versatec, therefore, the plotter must be free prior to initiating a plot. "Plotter busy" messages are generally received if the plotter is off-line or busy.

NAME

rsort — sort **cifout** files

SYNOPSIS

rsort [**infile**] [**-o outfile**] [**-x**] [**-y**] [**-l**]

DESCRIPTION

Rsort is a filter that sorts a **cifout** file. Stdin and stdout are the default input and output files. If the **infile** file name is specified, then the input is obtained from that file. The **-o** option indicates that the following argument is the name to be used for the output file.

The **-x**, **-y** and **-l** options specify which field of the data to sort on: the X coordinates, Y coordinates, or layer. The values are sorted into increasing order (minimum value first). The default is to sort by X coordinate which is needed by the **rplot** program. Only one option can be specified. Generally, the **-y** and **-l** options are used to completely sort a file to compare it to another file. The sort algorithm used is stable. Hence, two files that only differ because of line ordering will be identical after a full sort by layer, then Y, and then X coordinates. An example would be:

```
rsort test.sco -l | rsort -y | rsort -o test.sco.
```

FILES

/vlsi/lib/local/rsort

SEE ALSO

cifout(cad5), **rplot(cad1)**, **window(cad1)**

BUGS

NAME

sim — create a sim file for simulation with STAT, ESIM, or TSIM

SYNOPSIS

sim *file*

DESCRIPTION

Sim produces a simulation file (*.sim*) to be used in circuit simulation with the static checker (*stat*) or the event driven switch level simulator (*esim*). The input file name must have *.sym*, *.node*, and *.cap* files associated with it. The *.sym* file must have *vdd* and *gnd* nodes defined as a minimum. For example, to generate a *.sim* file for an extracted circuit with files of *test.sym*, *test.node*, and *test.cap*, first define the *vdd* and *gnd* nodes (as a minimum) in *test.sym* and then enter **sim test**. The result will be *test.sim*.

FILES

/vlsi/lib/local/extr/sim
/vlsi/lib/local/extr/gate1

SEE ALSO

extract(cad1), stat(cad1), esim(cad1)

BUGS

NAME

stat - the static checker

SYNOPSIS

stat file.sim [number][>file.stat]

DESCRIPTION

Stat performs a static (dc) analysis of *file.sim* produced by *extract* followed by *sim*. *Number* is the assumed number of threshold drops on the input pads. It is an optional input parameter with a default of 0.

Stat attempts to understand how transistors and nodes are used in the circuit. It summarizes this understanding in its output files. Two outputs are generated by *stat*. The standard error output (normally to the terminal) contains mainly counts of various items (node types, transistor types, etc.). The standard output (also to the terminal unless re-directed with >*file.stat*) contains detailed information about each potential error.

STDERR SUMMARY

- (1) A report of the number of nodes and transistors in the circuit.
This takes the form:

#nodes, #enhancement, #depletion, #intrinsic, #duplicates

Intrinsic transistors can be ignored since the current process does not build them. A *duplicate* transistor is a single logical transistor laid out physically as two or more transistors in parallel.

- (2) Transistor classifications:
[*de*] *gate source drain*
(*d*=depletion, *e*=enhancement)

d A A vdd	simple pullup
d A B vdd	part of a superbuffer
d A B C	ion-implant transistor
e A B C	typical transistor
e gnd A gnd	lightning arrestor
e A B gnd	pulldown
e A B vdd	unknown pullup

- (3) Input node count.
Any node N which contains a transistor of the form:

e gnd N gnd where length=2, width>=40

is considered to be an input node.

- (4) Bootstrap structure count.
The following is an example of a bootstrap structure:

d A B B bootstrap capacitor

- (5) Threshold drops on nodes.
Starting with the given input threshold drops (with *vdd*=0 and *gnd*=unknown), the information is propagated through transistors whose gate and source threshold drops are known, and drain unknown. Drain

node threshold drops are then calculated according to one of the following formulas:

depletion:	$\text{drain} = \max(\text{gate} - 3, \text{source})$
enhancement:	$\text{drain} = \max(\text{gate} + 1, \text{source})$

(6) Pullup node count.

Pullup nodes are classified into simple pullups, unknown pullups, and multiply pulled-up depending on the type of transistor(s) connected to the particular node in question. In the following structures:

d A B vdd	unknown depletion pullup transistor
e A B vdd	unknown enhancement pullup transistor

node B is marked as an unknown pullup node until a function has been found for it (such as part of a superbuffer). Unknown pullups are not necessary errors.

(7) Output node count.

Any node N which contains a transistor of the forms:

e B N vdd	where length=2 or 3, width>=280	large pullup
or		
e A N gnd	where length=2 or 3, width>=280	large pulldown

is considered to be an output up or down node.

(8) Pulldown transistor count.

A pulldown transistor is one that connects a strictly pulled-down node A to another node B. If node B is not pulled-up, then it is also strictly pulled-down, and can be used in finding other pulldown transistors.

(9) Pass transistor count.

(10) Logic gate count.

Where possible, logic gates are derived from transistor structures. Logic gates are: inverters, nors, and complicated gates (nand, xor, etc.).

(11) Superbuffer count.

(12) Ratio check and count.

All nodes that are simply pulled-up and connect to transistor gates are checked for the proper pu/pd ratio. Pulldown transistors with non-zero threshold drops on their gates are taken into account by making their lengths longer. Ratios that are < 4 or >= 5 are reported. The program cannot handle nodes with multiple simple pullups. When such a node is encountered, the message: **Program error in ratio** is displayed.

(13) Transistor error count.

Unknown depletion pullup transistors whose function cannot be determined are reported as:

unknown pullup transistors.

Enhancement transistors whose gate is *vdd* or *gnd*, whose source is *vdd* and drain *gnd* (or vice versa), whose source and drain are the same, or

whose gate is the same as its source or drain are reported as:

strange transistors.

Depletion pulldown transistors are reported as:

depletion mode pulldowns.

- (14) Node propagation error count.
 Four bits are associated with each node: **0**, **1**, **I**, **O**.
Gnd has the **0** bit set.
Vdd has the **1** bit set.
 All inputs have the **I** bit set.
 All outputs have the **O** bit set.
 The program propagates these bits through the circuit. In the end, nodes that do not have one or more of these bits set are counted and reported.

STDOUT MESSAGES

Most messages describe either a node or a transistor.
 The standard format for a node message is:

message: node (xpos,ypos)

The standard format for a transistor message is:

message: [de] gate source drain (xpos,ypos)

In the case of a pu/pd ratio message, the format is slightly more complicated:

r n (x,y) <ul x uw>: {<g [.,:;?] dl x dw> m}+

The message says: pu/pd ratio *r* is calculated for node *n* at position (*x,y*). Node *n* is pulled up with a pullup transistor of length *ul* and width of *uw*. Node *n* is pulled down to node *m* via a transistor whose gate is *g*, whose length is *dl* and width is *dw*, and *g* has one of 5 possible threshold drops [.,:;?] on it. One or more, {}, pulldown transistors can exist in the pulldown path, the last of which must have *m=gnd* (obviously).

The various threshold drops are denoted by:

	symbol	drop	effective pd resistance
1.	.	0.0	x1.0
2.	.	0.5	x1.5
3.	:	1.0	x2.0
4.	:	1.5	x2.5
5.	?	unknown	x infinity

Threshold drop changes the effective resistance of a pulldown transistor used in ratio calculation.

FILES

/vlsi/lib/local/extr/stat

SEE ALSO

extract(cad1),sim(cad1)

BUGS

Only if you don't believe what the program tells you.

It is recommended that *vdd*, *gnd*, *phi1*, and *phi2* be defined in the *file.sim* before subjecting it to abuse by *stat*.

NAME

tplot — plots a cifout file on a GIGI terminal

SYNOPSIS

tplot [options]...file...

DESCRIPTION

Tplot is a program that can be run from a GIGI terminal. It will produce a color plot of a cifout file. The layers plotted and their respective colors and dot patterns are

metal	blue
diffusion	green
polysilicon	red
implant	yellow
contact	magenta
glass/DRC error	white	— — —
unknown	cyan	???????

The processing can be modified by the following switches:

- i#** Use a scale factor of # lambdas per inch.
- nl** Causes the named layer, *l*, to be omitted from the plot. The layer can be one or more of: **c, d, g, i, m, p**.
- s#1.#2** Divides the chip into #2 strips and plots the #1'th strip.
- x#1.#2** Sets the minimum x to be plotted as #1 lambda and the maximum as #2 lambda.
- y#1.#2** Sets the minimum y to be plotted as #1 lambda and the maximum as #2 lambda.

If no indications of area to plot are given, *tplot* will scale the plot to best fit the terminal screen.

After the plot is complete, the terminal will go into the "position mode." In this mode, the terminal "arrow" keys can be used to move the graphics cursor to any desired position on the screen. If the SHIFT key is held down in conjunction with an "arrow" key, the cursor will move ten units at a time. Once the cursor has been moved to the desired position, a 'p' will cause the terminal to display the cursor position in lambdas. A 'q' will erase the screen and terminate the program.

FILES

/vlsi/lib/local/tplot

SEE ALSO

cifout(cad5)

BUGS

NAME

`unconvert` — converts an ASCII cifout file to binary form

SYNOPSIS

`unconvert` < *file*

DESCRIPTION

Unconvert takes an ASCII cifout file from standard input and converts it to a binary format sent to standard output.

FILES

/vlsi/lib/local/unconvert

SEE ALSO

`cifout(cad5)`, `convert(cad1)`

BUGS

NAME

window window a cifout file

SYNOPSIS

window [-x#1.#2] [-y#1.#2] [-l#] [-sn.m] [-nl]

DESCRIPTION

Window is a filter that converts a **cifout** file to raster coordinates preparatory to conversion to raster format. *Window* can convert any selected portion of the IC and also scale the resulting plot. The processing can be modified by the following switches:

- x#1.#2** set the minimum x to be plotted as #1, max x as #2 (in lambdas, either #1 or #2 can be omitted)
- y#1.#2** set minimum y as #1, maximum y as #2 (either #1 or #2 can be omitted)
- l#** sets scale factor to # lambdas per inch
- sn.m** plots strip *n* of *m* strips. This allows convenient plotting of IC's that are too large to fit onto a page. Note that *n* ranges from 1 to *m*. Strip 1 is the first strip (lower left corner), strip 2 the second, etc.
- nl** causes the named layer, *l*, to be omitted from the plot. The layer, *l*, may be one or more of **c,d,g,i,m,p** or **z**.
- b** Blockout. Causes the min to max x (and y) to be blocked out rather than plotted. Creates a box on the Z layer to show where the block was. Automatically sets the -u option. Note, the output file may be unsorted even if the input file was sorted.
- u** Unscaled. Causes the output coordinates to be unscaled to raster coordinates. This allows a file to be windowed more than once without the coordinates getting scaled to fit the Varian every time.
- d** Causes the object to be windowed to the size of a normal page for documentation (suppresses printing of comments). Will make the 10.5 inch dimension in either the x direction or the y direction to get the biggest possible plot.
- r** Causes the object to be windowed to the size of a bound page so that it may be bound (also suppresses printing of comments).
- gx.y** Allows you to specify a grid to be displayed on the plot. This inserts appropriate instructions into the *cifout* file to cause **rplot** to plot grid lines at a spacing of *x* lambda in the x direction and *y* lambda in the y direction. If *x* or *y* are omitted a default value of 5 lambda is used. If no arguments are specified, *window* will use the bounding box for min/max x/y and scale the plot to best fit the paper. The user can specify any subset of parameters that he wishes, and *window* will use the given information in conjunction with the information in the *cifout* file to determine the desired operation. For example, the command
window -x.53
 would use the bounding box information to determine xmin, ymin and ymax, but xmax would be set to 53 lambda.

FILES

/vlsi/lib/local/window

SEE ALSO

cifout(cad5), rsort(cad1), rplot(cad1)

BUGS

Not too much error checking for ridiculous arguments or duplicate arguments.

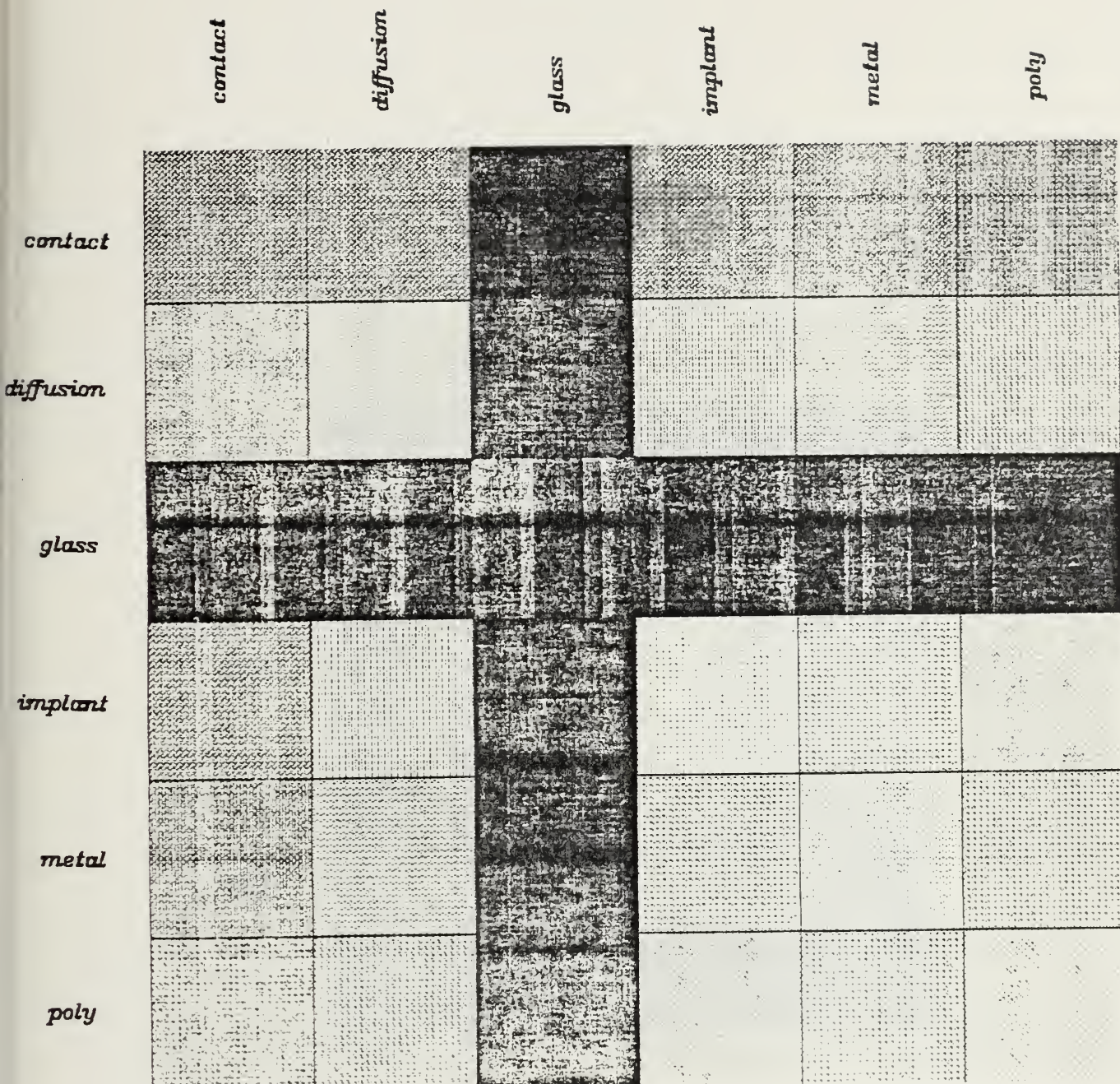


FIGURE 1 - CLL/RPLOT STIPPLE PATTERN

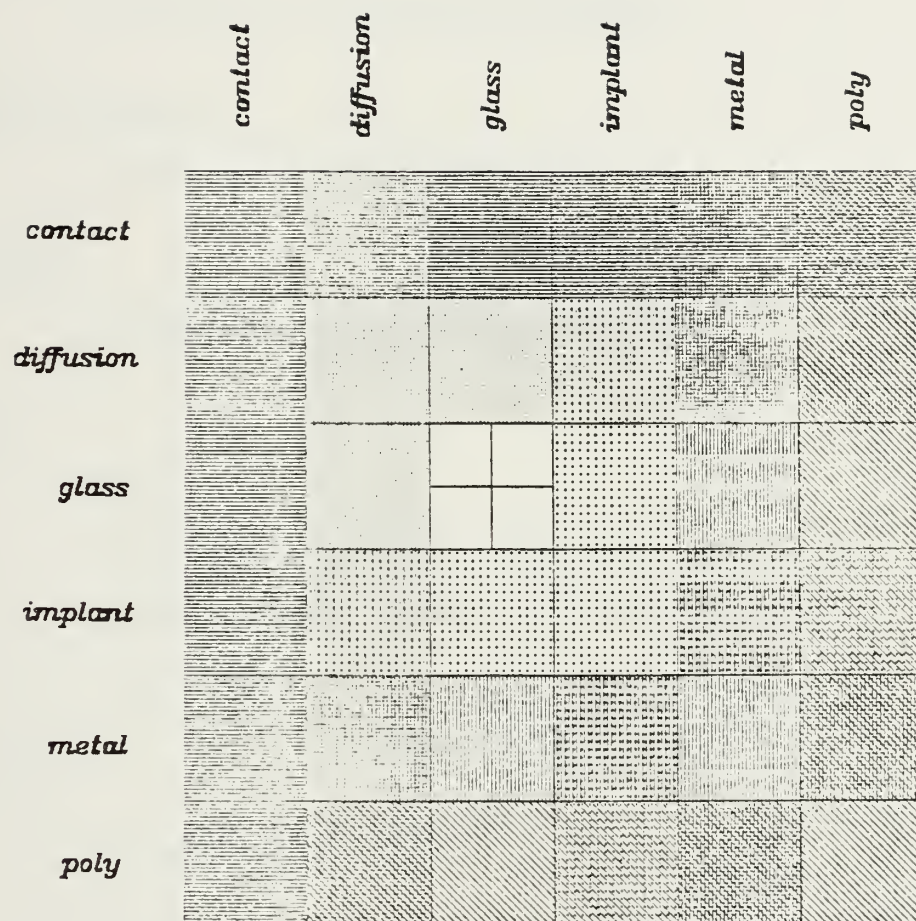


FIGURE 2 – NODE-PLOT STIPPLE PATTERN

APPENDIX C

SUMMARY OF CLL COMMANDS¹

COMMENTS

enclosed in /* */ ; commands do **NOT** nest.

SYMBOL DEFINITION

name [(cif# bounds llx,lly xlen,ylen)]

EXTERNAL

external *name* (cif# bounds llx,lly xlen,ylen)

LAYER

layer;

(metal, blue, red, diffusion, diff, green,

contact, cut, black, implant, yellow, glass,

metal2, poly2)

RECTANGLE

rect llx,lly xlen,ylen [*layer*];

or

r llx,lly xlen,ylen [*layer*];

VIA

via llx,lly [*layer*]

(poly or diffusion)

WIRE

Excerpt from Reference 5.

wire [*layer*] x,y *wirelist*;

or

w [*layer*] x,y *wirelist*;

(*wirelist* consists of one or more of:

u #, d #, r #, l #, w #, x #, y #, #.#, *layer*)

CALL

name(llx,lly *transformations*);

ITERATE

iterate nx,ny [xpitch,ypitch]

name(llx,lly *transformations*);

TRANSFORMATIONS

flip ud, flip lr, flip rl

rotate 0, rotate 3, rotate 6, rotate 9, rotate 12

FUNCTIONS

dx(*name*) dy(*name*) pwidth(*#ma*) print(*expr*)

DEFINES

```
#define symbol-name real-value
```

INCLUDES

```
#include "file-name"
```

CONDITIONAL

```
#ifdef x
```

```
...
```

```
#endif
```

```
or
```

```
#ifndef x
```

```
...
```

```
#endif
```

CLL RESERVED WORDS

black	blue	bounds	butt
cif	contact	cut	d
default	diff	diffusion	dx
dy	external	flip	glass
green	implant	iterate	l
lr	metal	metal2	poly
poly2	print	pwidth	r
rect	red	rl	rotate
u	ud	via	w
wire	x	y	yellow

STANDARD FILE STRUCTURE


```

#ifndef MYNAME
#define MYNAME

#include "/vlsi/lib/local/s_ext.cll"
#include "a"
#include "b"

external name(cif# bounds llx,lly xlen,ylen)

#define x 1
#define y (a-y+3)
#define z 7.5

symbol-name()
{
...
}

#endif

```


APPENDIX D

DESIGN FABRICATION

The following sections provide an overview of the procedure for using the DARPA Net to deliver a CIF file to MOSIS (Chapter 2 of this thesis) for fabrication.

A. INTRODUCTION TO THE DARPA NET

The DARPA Net is the computer link between the designer and MOSIS, where the CIF file is verified and forwarded for fabrication. Access to this net is controlled and will not be covered by this Appendix; however, once access has been obtained, the following material will be a guide to the user.

At present, the NPS VAX computer is not capable of linking to the DARPA Net. Until this connection is available, the user must use a remote terminal with modem capability. The phone numbers for the Net link are:

646-3150	(300 BAUD)
646-3158	(1200 BAUD)

Both are full duplex operation. After a link has been made, the DARPA Net can be established by pressing the terminal's CONTROL and Q keys simultaneously (<CTRL>Q). The terminal responds when the net has been opened and waits for the user to open a host computer tie.

Although there are several host computers capable of accessing the DARPA Net, the two most frequently used at NPS are ECLB and ISIE. To open the connection with ECLB, the user should type

@o 23<CR>

where <CR> is RETURN. To open the connection with ISIE, type

@o 1/52<CR>

In either case, the net responds to the open connection with information about the net and then issues the system prompt

@

1. Login/Logout

The command to log onto the net is

login *user-name* *password*<CR>

The computer responds with information about the account and then issues the system prompt again. The system now accepts valid commands.

Since the DARPA Net is a shared net, the response of the computer is generally slow. Be patient and don't attempt to confuse the computer with several commands while it is attempting to execute one. If at any time it appears that the link has been lost or the system is "locked-up," simply terminate the modem connection. The net closes the connection after a set amount of "idle time."

To properly log off of the link, type

logout<CR>

The system responds with a message confirming that it is closing the connection.

2. Help

To obtain a list of valid commands, type

?<CR>

The net also has a **HELP** function which provides information and usage for particular commands. To find out what commands are supported by **HELP**, type

help ?<CR>

The command

help *name*

gives information about the system command, *name*. The two commands that will be used most by the VLSI designer are **MSG** and **SENDMSG**.

3. **MSG**

The manual for **MSG** can be obtained with the command

help msg

(This is a long file and should be printed for user reference.) **MSG** will be used to read and send mail (messages or letters) within the DARPA Net and especially to MOSIS.

To determine if there is any mail that has not been examined, type

msg

The computer responds by indicating if any mail is stored and will give the message number and origin. It will terminate with the **MSG** prompt of

<-

To read a message, type

t *number*

where *number* is the message number (or range of numbers). For example, **t 52** causes message 52 to be displayed, while **t 52-60** causes messages 52 through 60 to be displayed consecutively.

To send a message while in the **MSG** function, type

s

The computer responds with

To (? for help):

The address of the the user to receive the message should now be entered. For example,

MOSIS@@USC-ISIF

Note that two "@" keys must be typed, while the terminal will type three of them. After a RETURN, the system responds with

cc (? for help):

This is a request for the address of a user who is to get a copy of the message. It is recommended that the designer put his address to get a copy of the transmitted message. After entering this address, the computer responds with

Subject:

The user then enters the subject of the message. The next input requested by the computer is

Message (? for help):

The text of the message can now be typed. While entering the message text, various editor commands are available. The commands are listed in the **SNDMSG** manual but the most commonly used ones are:

<CTRL>D	Retype text
<CTRL>H	Delete last character
<CTRL>U	Delete present line
<CTRL>Z	End of message text
<CTRL>N	Abort this message

After the message has been entered and the user has indicated that the end of the text has been reached (with <CTRL>Z), The computer will ask if the message should be sent (S) or placed in memory (Q). If the user responds with

S

the message will be sent after the addresses have been confirmed.

Any additional information on the DARPA Net should be obtained from the **HELP** function.

B. MOSIS

MOSIS is the link between the designer and the fabrication facilities. It provides information on the current schedule for the technologies that are being fabricated and also information concerning updates to these technologies (nMOS, cMOS, etc.). Although the *MOSIS USER'S MANUAL* [Ref. 8] provides a complete list of procedures for the fabrication process, this Appendix highlights the major points.

1. Obtaining Information

Since MOSIS has an automatic message processing system, all correspondence to it must be in standard format and identified with valid subject and

request lines. The format of the text for messages to MOSIS is:

```
REQUEST: Type-of-Request  
        Parameter line  
.  
.  
.  
REQUEST: END
```

The allowed entries for the REQUEST and Parameter lines are given in the User's Manual. To obtain this manual (along with other basic information), the following message should be sent:

```
TO: MOSIS@@USC-ISIF  
CC: User-Address  
SUBJECT: INFORMATION REQUEST  
  
REQUEST: INFORMATION  
        TOPIC: USER-MANUAL  
        TOPIC: GENERAL  
        TOPIC: TOPICS  
REQUEST: END
```

The GENERAL topic provides information on how to obtain authorization to use MOSIS and the TOPICS topic gives information on other topics relating to the MOSIS service.

The turn-around time for a request to MOSIS is generally less than one hour during working hours. Once these basic information sources have been received, the user will be able to request information on other areas (library, schedule, etc.).

2. Required Messages

Once authorization has been obtained to use the MOSIS service, the designer needs to initiate several messages in order to get a chip fabricated. All of these messages are documented in the *MOSIS USER'S MANUAL*. However, the messages that are absolutely required are requests for NEW PROJECT,

FABRICATE, and REPORT.

The NEW-PROJECT request has the form:

```
REQUEST: NEW-PROJECT
  D-NAME:                (name of designer)
  AFFILIATION:            (Navy)
  ACCOUNT:                (MOSIS account number)*
  D-PASSWORD:            (designer's password)
  NET-ADDRESS:            (designer's net address)
  MAILING-ADDRESS:        (designer's mailing address)
  P-NAME:                (project's name)
  P-PASSWORD:            (project's password)
  DESCRIPTION:            (short description of project)
  TECHNOLOGY:            (nMOS,cMOS,etc.)
  LAMBDA:                (requested lambda)
  MIN-LAMBDA:            (min accepted)
  MAX-LAMBDA:            (max accepted)
  PADS:                  (number of pads)
REQUEST: END
```

* Assigned by MOSIS after authorization has been granted.

MOSIS replies to this message with an approval (or disapproval) message which gives a project number.

The FABRICATE message can be used both to submit a CIF file and request that it be fabricated. It has the form:

```
REQUEST: FABRICATE
  ID:                    (project # assigned by MOSIS)
  P-PASSWORD:            (project password)
  SIZE:                  (length X width of project in microns)
  CIF:
  (insert final.cif here)
REQUEST: END
```

This is the minimum requirement for the message. Other information can be added if the designer feels that it is necessary. For example, the lambda used to calculate SIZE may be included. Additionally, if a check-sum was performed on the CIF file (See the following section.), it should be included in this message. MOSIS responds to this message with a "valid CIF" (or "not valid CIF") message.

If the "not valid CIF" message is received, the designer must retransmit his FABRICATE message.

The REPORT message should be sent after the chip has been received and tested. This provides feedback to MOSIS for their analysis of the fabrication of different technologies. It has the form:

```
REQUEST: REPORT
        ID:                (project number)
        P-PASSWORD:        (project password)
        REPORT:            (report of performance of
                           fabricated project)
REQUEST: END
```

3. Cksum

MOSIS provides the software for performing a "check-sum" on a CIF file which is used to validate that file. Check-sum gives an output which is a unique count of the input CIF file that can be used to verify correct transmission of the file over a data link. This software has been installed on the VAX and should be used by the designer. If the command

cksum final.cif

is issued, the computer responds with

```
CIF-CHECKSUM= number1
BYTE-COUNT= number2
```

The check-sum can be included in the FABRICATE request, while the byte-count is for the designer's information only. Upon receipt of the CIF file, MOSIS computes a checksum and reports its value in the "valid CIF" message. The designer should verify that this check-sum is identical to *number1*.

APPENDIX E
FILES AND PROGRAMS FOR THESIS PROJECT

pla1

CIF# 950;

in: A0 B0 A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A10 B10 A11;

in: B11 A12 B12 A13 B13 A14 B14 A15 B15 CIN;

out: G140 G130 G120 G110 G100 G90 G80 G70 G60 G50 G40 G30 G20 G10 G00 P150;

out: P140 P130 P120 P110 P100 P90 P80 P70 P60 P50 P40 P30 P20 P10 P00 G21;

out: G11 G01 P21 P11 P01 COUT;

G140=A14&B14;

G130=A13&B13;

G120=A12&B12;

G110=A11&B11;

G100=A10&B10;

G90 =A9&B9;

G80 =A8&B8;

G70 =A7&B7;

G60 =A6&B6;

G50 =A5&B5;

G40 =A4&B4;

G30 =A3&B3;

G20 =A2&B2;

G10 =A1&B1;

G00 =A0&B0;

P150=A15'&B15+A15&B15';

P140=A14'&B14+A14&B14';

P130=A13'&B13+A13&B13';

P120=A12'&B12+A12&B12';

P110=A11'&B11+A11&B11';

P100=A10'&B10+A10&B10';

P90 =A9'&B9+A9&B9';

P80 =A8'&B8+A8&B8';

P70 =A7'&B7+A7&B7';

P60 =A6'&B6+A6&B6';

P50 =A5'&B5+A5&B5';

P40 =A4'&B4+A4&B4';

P30 =A3'&B3+A3&B3';

P20 =A2'&B2+A2&B2';

P10 =A1'&B1+A1&B1';

P00 =A0'&B0+A0&B0';

PLA1 EQUATIONS CONTINUED ON NEXT PAGE

pla2

```
CIF# 951;
in: G140 G130 G120 G110 G100 G90 G80 G70 G60 G50 G40 G30 G20 G10 G00;
in: P150 P140 P130 P120 P110 P100 P90 P80 P70 P60 P50 P40 P30 P20 P10;
in: P00 G21 G11 G01 P21 P11 P01 CIN;
out: OP15 OP14 OP13 OP12 OP11 OP10 OP9 OP8 OP7 OP6 OP5 OP4 OP3 OP2 OP1;
out: OP0 OG14 OG13 OG12 OG11 OG10 OG9 OG8 OG7 OG6 OG5 OG4 OG3 OG2 OG1;
out: OG0 C11 C7 C3 COUT;
C3=G01+CIN&P01;
C7=G11+G01&P11+CIN&P01&P11;
C11=G21+G11&P21+G01&P11&P21+CIN&P01&P11&P21;
OG0=G00;
OG1=G10;
OG2=G20;
OG3=G30;
OG4=G40;
OG5=G50;
OG6=G60;
OG7=G70;
OG8=G80;
OG9=G90;
OG10=G100;
OG11=G110;
OG12=G120;
OG13=G130;
OG14=G140;
OP0=P00;
OP1=P10;
OP2=P20;
OP3=P30;
OP4=P40;
OP5=P50;
OP6=P60;
OP7=P70;
OP8=P80;
OP9=P90;
OP10=P100;
OP11=P110;
OP12=P120;
OP13=P130;
OP14=P140;
OP15=P150;
COUT=CIN;
```


pla3

```
CIF# 952;
in: P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0;
in: G14 G13 G12 G11 G10 G9 G8 G7 G6 G5 G4 G3 G2 G1 G0 C11 C7 C3 CIN;
out: COUT OC0 OC1 OC2 OC3 OC4 OC5 OC6 OC7 OC8 OC9 OC10 OC11 OC12 OC13;
out: OC14 OP0 OP1 OP2 OP3 OP4 OP5 OP6 OP7 OP8 OP9 OP10 OP11 OP12;
out: OP13 OP14 OP15;
COUT=CIN;
OC0=G0+CIN&P0;
OC1=G1+G0&P1+CIN&P0&P1;
OC2=G2+G1&P2+G0&P1&P2+CIN&P0&P1&P2;
OC3=C3;
OC4=G4+C3&P4;
OC5=G5+G4&P5+C3&P4&P5;
OC6=G6+G5&P6+G4&P5&P6+C3&P4&P5&P6;
OC7=C7;
OC8=G8+C7&P8;
OC9=G9+G8&P9+C7&P8&P9;
OC10=G10+G9&P10+G8&P9&P10+C7&P8&P9&P10;
OC11=C11;
OC12=G12+C11&P12;
OC13=G13+G12&P13+C11&P12&P13;
OC14=G14+G13&P14+G12&P13&P14+C11&P12&P13&P14;
OP0=P0;
OP1=P1;
OP2=P2;
OP3=P3;
OP4=P4;
OP5=P5;
OP6=P6;
OP7=P7;
OP8=P8;
OP9=P9;
OP10=P10;
OP11=P11;
OP12=P12;
OP13=P13;
OP14=P14;
OP15=P15;
```


pla4

CIF# 953;

in: P15 P14 P13 P12 P11 P10 P09 P08 P07 P06 P05 P04 P03 P02 P01 P00;

in: C14 C13 C12 C11 C10 C09 C08 C07 C06 C05 C04 C03 C02 C01 C00 CIN;

out: S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15;

S0=CIN'&P00+CIN&P00';

S1=C00'&P01+C00&P01';

S2=C01'&P02+C01&P02';

S3=C02'&P03+C02&P03';

S4=C03'&P04+C03&P04';

S5=C04'&P05+C04&P05';

S6=C05'&P06+C05&P06';

S7=C06'&P07+C06&P07';

S8=C07'&P08+C07&P08';

S9=C08'&P09+C08&P09';

S10=C09'&P10+C09&P10';

S11=C10'&P11+C10&P11';

S12=C11'&P12+C11&P12';

S13=C12'&P13+C12&P13';

S14=C13'&P14+C13&P14';

S15=C14'&P15+C14&P15';

stage1.cll

```
1  /* include cell library */
2  # include "/vlsi/lib/local/s_ext.cll"
3  /* define pla1 cif file*/
4  external pla1(cif 950 bounds --15,0 868,1151)
5  /*place pla1 */
6  stage1()
7  {
8      pla1(0,123);
9      iterate 33,1
10         Afterburner(16,58);
11         iterate 33,1
12             PlaclockIn(15,0);
13             iterate 19,1
14                 PlaclockOut (556,70);
15 }
```


stage2.cll

```
/*include cell library */
# include "/vlsi/lib/local/s_ext.cll"
/* define pla2 cif file */
external pla2(cif 951 bounds --15,0 924,352)
/* place pla 2*/
stage2()

{
  pla2(0,15);
  iterate 18,1
    PlaPullup (638,4 rotate 9);
  iterate 38,1
    Afterburner (10,361 rotate 6);
  iterate 38,1
    PlaClockIn (11,426 rotate 6);
  iterate 18,1
    PlaClockOut (638,359 rotate 6);
}
```


stage3.cll

```
/* include cell library */
# include "/vlsi/lib/local/s_ext.cll"
/* define pla3 cif file */
external pla3(cif 952 bounds --15,0 852,464)
/* place pla3 */
stage3()
{
    pla3(0,123);
    iterate 35,1
        Afterburner(16,58);
    iterate 35,1
        PlaClockIn (15,0);
    iterate 16,1
        PlaClockOut (590,579 rotate 6);
    iterate 16,1
        PlaPullup (590,112 rotate 9);
}
```


stage4.cll

```
/* include cell library */
#include "/vlsi/lib/local/s_ext.cll"
/* define pla4 cif file */
external pla4(cif 953 bounds --15,0 676,264)
/* place pla4 */
stage4()
{
    pla4(0,117 rotate 6);
    iterate 32,1
        Afterburner(154,58);
    iterate 32,1
        PlaClockIn(153,0);
    iterate 8,1
        PlaClockOut(8,377 rotate 6);
    iterate 8,1
        PlaPullup (8,110 rotate 9);
}
```


stage5.cll

```

/* this stage will develop the input-output pads */
/* and will be combined with stages 1 thru 4 */
/* include cell library */
#include "/vlsi/lib/local/s_ext.cll"
/* place input/output pads */
stage5()
{
    /* lower edge pads */
    iterate 14,1 150,0
        NIn8 (225,0);
    /* left edge pads */
    iterate 1,14 0,150
        NIn8 (0,225 rotate 3);
    /* top edge pads */
    iterate 7,1 150,0
        NIn8 (225,2568 rotate 6);
    iterate 7,1 150,0
        NOut8 (1275,2555 rotate 6);
    /* right edge pads */
    iterate 1,9 0,150
        NOut8 (2355,975 rotate 9);
        NVdd (2420,825 rotate 9);
        NGnd (2394,675 rotate 9);
}

```



```

1  /*include cell library*/
2  # include "/vlsi/lib/local/s_ext.cll"
3  # include "stage1.cll"
4  # include "stage2.cll"
5  # include "stage3.cll"
6  # include "stage4.cll"
7  # include "stage5.cll"
8  # include "designer.cll"
9  main()
10 {
11     stage1(340,920);
12     stage2(882,240);
13     stage3(1270,1160);
14     stage4(1454,2000);
15     stage5(0,0);
16     /* pla1 out to pla2 in */
17     wire poly 899,990 y 985 metal w 3 y 730 diff y 720;
18     wire poly 907,990 y 985 metal w 3 y 730 diff y 724 x 915 y 720;
19     wire poly 915,990 y 985 metal w 3 y 730 x 931 diff y 720;
20     wire poly 923,990 y 985 metal w 3 y 737 x 947 y 730 diff y 720;
21     wire poly 931,990 y 985 metal w 3 y 744 x 963 y 730 diff y 720;
22     wire poly 939,990 y 985 metal w 3 y 751 x 979 y 730 diff y 720;
23     wire poly 947,990 y 985 metal w 3 y 758 x 995 y 730 diff y 720;
24     wire poly 955,990 y 985 metal w 3 y 765 x 1011 y 730 diff y 720;
25     wire poly 963,990 y 985 metal w 3 y 772 x 1027 y 730 diff y 720;
26     wire poly 971,990 y 985 metal w 3 y 779 x 1043 y 730 diff y 720;
27     wire poly 979,990 y 985 metal w 3 y 786 x 1059 y 730 diff y 720;
28     wire poly 987,990 y 985 metal w 3 y 793 x 1075 y 730 diff y 720;
29     wire poly 995,990 y 985 metal w 3 y 800 x 1091 y 730 diff y 720;
30     wire poly 1003,990 y 985 metal w 3 y 807 x 1107 y 730 diff y 720;
31     wire poly 1011,990 y 985 metal w 3 y 814 x 1123 y 730 diff y 720;
32     wire poly 1019,990 y 985 metal w 3 y 821 x 1139 y 730 diff y 720;
33     wire poly 1027,990 y 985 metal w 3 y 828 x 1155 y 730 diff y 720;
34     wire poly 1035,990 y 985 metal w 3 y 835 x 1171 y 730 diff y 720;
35     wire poly 1043,990 y 985 metal w 3 y 842 x 1187 y 730 diff y 720;
36     wire poly 1051,990 y 985 metal w 3 y 849 x 1203 y 730 diff y 720;
37     wire poly 1059,990 y 985 metal w 3 y 856 x 1219 y 730 diff y 720;
38     wire poly 1067,990 y 985 metal w 3 y 863 x 1235 y 730 diff y 720;
39     wire poly 1075,990 y 985 metal w 3 y 870 x 1251 y 730 diff y 720;
40     wire poly 1083,990 y 985 metal w 3 y 877 x 1267 y 730 diff y 720;
41     wire poly 1091,990 y 985 metal w 3 y 884 x 1283 y 730 diff y 720;
42     wire poly 1099,990 y 985 metal w 3 y 891 x 1299 y 730 diff y 720;
43     wire poly 1107,990 y 985 metal w 3 y 898 x 1315 y 730 diff y 720;
44     wire poly 1115,990 y 985 metal w 3 y 905 x 1331 y 730 diff y 720;
45     wire poly 1123,990 y 985 metal w 3 y 912 x 1347 y 730 diff y 720;
46     wire poly 1131,990 y 985 metal w 3 y 919 x 1363 y 730 diff y 720;
47     wire poly 1139,990 y 985 metal w 3 y 926 x 1379 y 730 diff y 720;
48     wire poly 1147,990 y 985 metal w 3 y 933 x 1395 y 730 diff y 720;
49     wire poly 1155,990 y 985 metal w 3 y 940 x 1411 y 730 diff y 720;
50     wire poly 1163,990 y 985 metal w 3 y 947 x 1427 y 730 diff y 720;
51     wire poly 1171,990 y 985 metal w 3 y 954 x 1443 y 730 diff y 720;
52     wire poly 1179,990 y 985 metal w 3 y 961 x 1459 y 730 diff y 720;

```



```

53      wire poly 1187,990 y 985 metal w 3 y 968 x 1475 y 730 diff y 720;
54      wire poly 1195,990 y 985 metal w 3 y 975 x 1491 y 730 diff y 720;
55      /* pla2 out to pla3 in */
56      wire poly 1797,652 y 657 metal w 3 y 1120 x 1839 y 1155 diff y 1160;
57      wire poly 1789,652 y 657 metal w 3 y 1126 x 1823 y 1155 diff y 1160;
58      wire poly 1781,652 y 657 metal w 3 y 1132 x 1807 y 1155 diff y 1160;
59      wire poly 1773,652 y 657 metal w 3 y 1138 x 1791 y 1155 diff y 1160;
60      wire poly 1765,652 y 657 metal w 3 y 1144 x 1775 y 1155 diff y 1160;
61      wire poly 1757,652 y 657 metal w 3 y 1150 x 1759 y 1155 diff y 1160;
62      wire poly 1749,652 y 657 metal w 3 y 1150 x 1743 y 1155 diff y 1160;
63      wire poly 1741,652 y 657 metal w 3 y 1144 x 1727 y 1155 diff y 1160;
64      wire poly 1733,652 y 657 metal w 3 y 1138 x 1711 y 1155 diff y 1160;
65      wire poly 1725,652 y 657 metal w 3 y 1132 x 1695 y 1155 diff y 1160;
66      wire poly 1717,652 y 657 metal w 3 y 1126 x 1679 y 1155 diff y 1160;
67      wire poly 1709,652 y 657 metal w 3 y 1120 x 1663 y 1155 diff y 1160;
68      wire poly 1701,652 y 657 metal w 3 y 1114 x 1647 y 1155 diff y 1160;
69      wire poly 1693,652 y 657 metal w 3 y 1108 x 1631 y 1155 diff y 1160;
70      wire poly 1685,652 y 657 metal w 3 y 1102 x 1615 y 1155 diff y 1160;
71      wire poly 1677,652 y 657 metal w 3 y 1096 x 1599 y 1155 diff y 1160;
72      wire poly 1669,652 y 657 metal w 3 y 1090 x 1583 y 1155 diff y 1160;
73      wire poly 1661,652 y 657 metal w 3 y 1084 x 1567 y 1155 diff y 1160;
74      wire poly 1653,652 y 657 metal w 3 y 1078 x 1551 y 1155 diff y 1160;
75      wire poly 1645,652 y 657 metal w 3 y 1072 x 1535 y 1155 diff y 1160;
76      wire poly 1637,652 y 657 metal w 3 y 1066 x 1519 y 1155 diff y 1160;
77      wire poly 1629,652 y 657 metal w 3 y 1060 x 1503 y 1155 diff y 1160;
78      wire poly 1621,652 y 657 metal w 3 y 1054 x 1487 y 1155 diff y 1160;
79      wire poly 1613,652 y 657 metal w 3 y 1048 x 1471 y 1155 diff y 1160;
80      wire poly 1605,652 y 657 metal w 3 y 1042 x 1455 y 1155 diff y 1160;
81      wire poly 1597,652 y 657 metal w 3 y 1036 x 1439 y 1155 diff y 1160;
82      wire poly 1589,652 y 657 metal w 3 y 1030 x 1423 y 1155 diff y 1160;
83      wire poly 1581,652 y 657 metal w 3 y 1024 x 1407 y 1155 diff y 1160;
84      wire poly 1573,652 y 657 metal w 3 y 1018 x 1391 y 1155 diff y 1160;
85      wire poly 1565,652 y 657 metal w 3 y 1012 x 1375 y 1155 diff y 1160;
86      wire poly 1557,652 y 657 metal w 3 y 1006 x 1359 y 1155 diff y 1160;
87      wire poly 1549,652 y 657 metal w 3 y 1000 x 1343 y 1155 diff y 1160;
88      wire poly 1541,652 y 657 metal w 3 y 994 x 1327 y 1155 diff y 1160;
89      wire poly 1533,652 y 657 metal w 3 y 988 x 1311 y 1155 diff y 1160;
90      wire poly 1525,652 y 657 metal w 3 y 982 x 1295 y 1155 diff y 1160;
91      /* pla3 out to pla4 in */
92      wire poly 2113,1796 y 1800 metal w 3 y 1995 diff y 2000;
93      wire poly 2105,1796 y 1800 metal w 3 y 1990 x 2097 y 1995 diff y 2000;
94      wire poly 2097,1796 y 1800 metal w 3 y 1984 x 2081 y 1995 diff y 2000;
95      wire poly 2089,1796 y 1800 metal w 3 y 1978 x 2065 y 1995 diff y 2000;
96      wire poly 2081,1796 y 1800 metal w 3 y 1972 x 2049 y 1995 diff y 2000;
97      wire poly 2073,1796 y 1800 metal w 3 y 1966 x 2033 y 1995 diff y 2000;
98      wire poly 2065,1796 y 1800 metal w 3 y 1960 x 2017 y 1995 diff y 2000;
99      wire poly 2057,1796 y 1800 metal w 3 y 1954 x 2001 y 1995 diff y 2000;
100     wire poly 2049,1796 y 1800 metal w 3 y 1948 x 1985 y 1995 diff y 2000;
101     wire poly 2041,1796 y 1800 metal w 3 y 1942 x 1969 y 1995 diff y 2000;
102     wire poly 2033,1796 y 1800 metal w 3 y 1936 x 1953 y 1995 diff y 2000;
103     wire poly 2025,1796 y 1800 metal w 3 y 1930 x 1937 y 1995 diff y 2000;
104     wire poly 2017,1796 y 1800 metal w 3 y 1924 x 1921 y 1995 diff y 2000;
105     wire poly 2009,1796 y 1800 metal w 3 y 1918 x 1905 y 1995 diff y 2000;
106     wire poly 2001,1796 y 1800 metal w 3 y 1912 x 1889 y 1995 diff y 2000;
107     wire poly 1993,1796 y 1800 metal w 3 y 1906 x 1873 y 1995 diff y 2000;

```



```

108     wire poly 1985,1796 y 1800 metal w 3 y 1900 x 1857 y 1995 diff y 2000;
109     wire poly 1977,1796 y 1800 metal w 3 y 1894 x 1841 y 1995 diff y 2000;
110     wire poly 1969,1796 y 1800 metal w 3 y 1888 x 1825 y 1995 diff y 2000;
111     wire poly 1961,1796 y 1800 metal w 3 y 1882 x 1809 y 1995 diff y 2000;
112     wire poly 1953,1796 y 1800 metal w 3 y 1876 x 1793 y 1995 diff y 2000;
113     wire poly 1945,1796 y 1800 metal w 3 y 1870 x 1777 y 1995 diff y 2000;
114     wire poly 1937,1796 y 1800 metal w 3 y 1864 x 1761 y 1995 diff y 2000;
115     wire poly 1929,1796 y 1800 metal w 3 y 1858 x 1745 y 1995 diff y 2000;
116     wire poly 1921,1796 y 1800 metal w 3 y 1852 x 1729 y 1995 diff y 2000;
117     wire poly 1913,1796 y 1800 metal w 3 y 1846 x 1713 y 1995 diff y 2000;
118     wire poly 1905,1796 y 1800 metal w 3 y 1840 x 1697 y 1995 diff y 2000;
119     wire poly 1897,1796 y 1800 metal w 3 y 1834 x 1681 y 1995 diff y 2000;
120     wire poly 1889,1796 y 1800 metal w 3 y 1828 x 1665 y 1995 diff y 2000;
121     wire poly 1881,1796 y 1800 metal w 3 y 1822 x 1649 y 1995 diff y 2000;
122     wire poly 1873,1796 y 1800 metal w 3 y 1816 x 1633 y 1995 diff y 2000;
123     wire poly 1865,1796 y 1800 metal w 3 y 1810 x 1617 y 1995 diff y 2000;
124     /* pla1 vdd & gnd interconnects */
125     wire metal 1205,1043 w 4 y 1028 x 1200;
126     wire metal 1200,1000 w 4 x 1220 y 1196 x 1285;
127     wire metal 893,1043 w 4 y 1040 x 884;
128     wire metal 896,1028 w 4 x 890 y 1040;
129     wire metal 890,1028 w 4 y 975 x 883;
130     wire metal 342,1043 w 4 y 956 x 355;
131     wire metal 342,1001 w 4 x 356;
132     /* pla2 vdd & gnd interconnects */
133     wire metal 1811,595 w 4 y 614 x 1808;
134     wire metal 1520,614 w 4 x 1510 y 665 x 1501;
135     wire metal 1515,614 w 4 y 603;
136     wire metal 1520,242 w 4 x 884 y 251;
137     wire metal 884,595 w 4 y 639 x 892;
138     wire metal 884,639 w 4 y 684 x 893;
139     /* pla3 vdd & gnd interconnects */
140     wire metal 1860,1786 w 4 x 1272 y 1739;
141     wire metal 1272,1283 w 4 y 1240 x 1286;
142     wire metal 1272,1240 w 4 y 1196 x 1285;
143     wire metal 2119,1739 w 4 y 1758 x 2116;
144     wire metal 1860,1758 w 4 x 1855 y 1747;
145     wire metal 1855,1283 w 4 y 1280 x 1846;
146     wire metal 1855,1280 w 4 y 1215 x 1845;
147     wire metal 1845,1196 w 4 x 2118 y 1276;
148     /* pla4 vdd & gnd interconnects */
149     wire metal 1590,2424 w 4 x 2128 y 2381;
150     wire metal 2128,2125 w 4 y 2081 x 2120;
151     wire metal 2128,2081 w 4 y 2036 x 2119;
152     wire metal 1607,2036 w 4 x 1457 y 2112 x 1462;
153     wire metal 1457,2381 w 4 y 2396 x 1462;
154     wire metal 1590,2396 w 4 x 1593 y 2381;
155     wire metal 1600,2118 w 4 y 2055 x 1607;
156     /* end vdd & gnd interconnects for the pla's */
157     /* bonding pads in to pla1 inputs */
158     wire metal 2270,132 y 220 x 877 y 910 diff y 920;
159     wire metal 2120,132 y 212 x 861 y 910 diff y 920;
160     wire metal 1970,132 y 204 x 845 y 910 diff y 920;
161     wire metal 1820,132 y 196 x 829 y 910 diff y 920;
162     wire metal 1670,132 y 188 x 813 y 910 diff y 920;

```



```

163      wire metal 1520,132 y 180 x 797 y 910 diff y 920;
164      wire metal 1370,132 y 172 x 781 y 910 diff y 920;
165      wire metal 1220,132 y 164 x 765 y 910 diff y 920;
166      wire metal 1070,132 y 156 x 749 y 910 diff y 920;
167      wire metal 920,132 y 148 x 733 y 910 diff y 920;
168      wire metal 770,132 y 140 x 717 y 910 diff y 920;
169      wire metal 620,132 y 140 x 701 y 910 diff y 920;
170      wire metal 470,132 y 148 x 685 y 910 diff y 920;
171      wire metal 320,132 y 156 x 669 y 910 diff y 920;
172      wire metal 132,230 x 653 y 910 diff y 920;
173      wire metal 132,380 x 637 y 910 diff y 920;
174      wire metal 132,530 x 621 y 910 diff y 920;
175      wire metal 132,680 x 605 y 910 diff y 920;
176      wire metal 132,830 x 180 y 688 x 589 y 910 diff y 920;
177      wire metal 132,980 x 188 y 696 x 573 y 910 diff y 920;
178      wire metal 132,1130 x 196 y 704 x 557 y 910 diff y 920;
179      wire metal 132,1280 x 204 y 712 x 541 y 910 diff y 920;
180      wire metal 132,1430 x 212 y 720 x 525 y 910 diff y 920;
181      wire metal 132,1580 x 220 y 728 x 509 y 910 diff y 920;
182      wire metal 132,1730 x 228 y 736 x 493 y 910 diff y 920;
183      wire metal 132,1880 x 236 y 744 x 477 y 910 diff y 920;
184      wire metal 132,2030 x 244 y 752 x 461 y 910 diff y 920;
185      wire metal 132,2180 x 252 y 760 x 445 y 910 diff y 920;
186      wire metal 230,2568 y 2500 x 260 y 768 x 429 y 910 diff y 920;
187      wire metal 380,2568 y 2492 x 268 y 776 x 413 y 910 diff y 920;
188      wire metal 530,2568 y 2484 x 276 y 784 x 397 y 910 diff y 920;
189      wire metal 680,2568 y 2476 x 284 y 792 x 381 y 910 diff y 920;
190      wire metal 830,2568 y 2468 x 292 y 800 x 365 y 910 diff y 920;
191  /* end pads in to pla1 inputs */
192  /* phi 1 to pla1 & pla2 */
193      wire metal 980,2568 y 2460 x 300 y 922 poly x 355;
194      wire poly 882,922 w 2 x 888 y 718 x 893;
195  /* phi 1 to pla4 */
196      wire metal 980,2460 x 1220 y 2240 diff w 3 y 2190 metal x 1260 y 2002
197      poly x 1607;
198  /*phi 1 to pla3 from pla1 & pla2 */
199      wire metal 1260,2002 y 1220 poly y 1162 x 1288;
200  /* phi 2 pla1 */
201      wire metal 1130,2568 y 2470 x 1240 y 2220 diff w 3 y 2180 metal y 1220 poly
202      y 1041 x 1200;
203  /* phi 2 pla4 */
204      wire metal 1240,2383 x 1380 poly x 1462;
205  /* phi 2 pla2 & pla3 */
206      wire metal 1240,2440 w 3 x 2140 y 1745 poly x 2116;
207      wire metal 2140,1745 w 3 y 1120 x 1900 y 601 poly x 1803;
208  /* end clock distribution */
209  /* pla4 outputs to output bonding pads */
210      wire poly 1467,2434 y 2520 x 1325 y 2555;
211      wire poly 1475,2434 y 2555;
212      wire poly 1483,2434 y 2550 x 1625 y 2555;
213      wire poly 1491,2434 y 2545 x 1775 y 2555;
214      wire poly 1499,2434 y 2540 x 1925 y 2555;
215      wire poly 1507,2434 y 2532 metal w 3 x 2075 poly y 2555;
216      wire poly 1515,2434 y 2524 metal w 3 x 2225 poly y 2555;
217      wire poly 1523,2434 y 2516 metal w 3 x 2300 y 2225 x 2340 poly x 2355;

```



```

218      wire poly 1531,2434 y 2508 metal w 3 x 2290 y 2075 x 2340 poly x 2355;
219      wire poly 1539,2434 y 2500 metal w 3 x 2280 y 1925 x 2340 poly x 2355;
220      wire poly 1547,2434 y 2492 metal w 3 x 2270 y 1775 x 2340 poly x 2355;
221      wire poly 1555,2434 y 2484 metal w 3 x 2260 y 1625 x 2340 poly x 2355;
222      wire poly 1563,2434 y 2476 metal w 3 x 2250 y 1475 x 2340 poly x 2355;
223      wire poly 1571,2434 y 2468 metal w 3 x 2240 y 1325 x 2340 poly x 2355;
224      wire poly 1579,2434 y 2460 metal w 3 x 2230 y 1175 x 2340 poly x 2355;
225      wire poly 1587,2434 y 2452 metal w 3 x 2220 y 1025 x 2340 poly x 2355;
226      /* end pla4 output wire runs to the output bonding pads */
227      /* connect pad gnd & vdd */
228          wire metal 4,0 w 8 y 2700;
229          wire metal 0,2696 w 8 x 2500;
230          wire metal 2496,2700 w 8 y 0;
231          wire metal 0,4 w 8 x 2500;
232          wire metal 98,90 w 16 y 2610;
233          wire metal 90,2602 w 16 x 2410;
234          wire metal 2402,2610 w 16 y 90;
235          wire metal 90,98 w 16 x 2410;
236      /* vdd & gnd connects for pla's */
237      /* vdd for pla2 */
238          wire metal 2500,242 w 8 x 2450 diff w 8 x 2320 metal w 8 x 1860 w 4
239              x 1808;
240          wire metal 1860,242 w 4 y 642 x 1807;
241      /* gnd connect for pla3 */
242          wire metal 2402,1700 w 8 x 2270 diff w 8 x 2130 metal w 4 y 1758 x 2120;
243      /* vdd & gnd connect for pla4 */
244          wire metal 1260,2700 w 8 y 2640 diff w 8 y 2424 metal w 4 x 1463;
245          wire metal 1400,2602 w 8 y 2460 diff w 8 y 2396 metal w 4 x 1458;
246      /* pla3 connect vdd from pla4 */
247          wire metal 1457,2036 w 4 y 1786;
248      /* xtra vdd to pla3 */
249          wire metal 2500,1250 w 8 x 2430 diff w 8 x 2118;
250      /* pla2 gnd connect */
251          wire metal 2402,614 w 8 x 1920 diff w 8 x 1840 metal w 4 x 1811;
252      /* pla1 vdd connect */
253          wire metal 940,2700 w 8 y 2640 diff w 8 y 2440 metal w 8 y 2240 x 820 y 2191;
254      /* pla1 gnd connect */
255          wire metal 1400,2396 w 8 y 2202 x 1205 w 4 y 2175;
256      /* xtra vdd to pla1 */
257          wire metal 0,1100 w 8 x 70 diff w 8 x 320 metal w 8 x 342;
258      /* xtra gnd to pla1 */
259          wire metal 98,950 w 8 x 170 diff w 8 x 320 y 975 w 4 x 350
260          metal w 4 x 360;

261      /* put identification */
262          designer(1980,700);

263      }

```


designer.cll

```
/*generate a signature for the project*/
```

```
designer()  
{
```

```
poly;
```

```
/* HAUENSTEIN */
```

```
wire 0,0 u 20;  
wire 0,10 r 20;  
wire 20,0 u 20;  
wire 30,0 u 20 r 20 d 20;  
wire 30,10 r 20;  
wire 60,20 d 20 r 20 u 20;  
wire 110,0 l 20 u 20 r 15;  
wire 90,10 r 10;  
wire 120,0 u 20;  
wire 140,0 u 20;  
wire 125,13 u 6;  
wire 130,7 u 6;  
wire 135,1 u 6;  
wire 150,0 r 20 u 10 l 20 u 10 r 20;  
wire 190,0 u 20;  
wire 180,20 r 20;  
wire 230,0 l 20 u 20 r 15;  
wire 210,10 r 10;  
wire 240,0 r 20;  
wire 240,20 r 20;  
wire 250,0 u 20;  
wire 270,0 u 20;  
wire 290,0 u 20;  
wire 275,13 u 6;  
wire 280,7 u 6;  
wire 285,1 u 6;
```

```
/* CONRADI */
```

```
wire 20,40 l 20 u 20 r 15;  
wire 30,40 u 20 r 20 d 20 l 21;  
wire 60,40 u 20;  
wire 80,40 u 20;  
wire 65,53 u 6;  
wire 70,47 u 6;  
wire 75,41 u 6;  
wire 90,40 u 20 r 20 d 10 l 21;  
wire 105,50 d 5;  
wire 110,40 u 5;  
wire 120,40 u 20 r 20 d 20;  
wire 120,50 r 20;  
wire 150,40 u 20 r 20 d 20 l 21;  
wire 152,40 u 20;
```


wire 180,40 r 20;
wire 180,60 r 20;
wire 190,40 u 20;

/* NAVY PGS '83' */

wire 0,80 u 20;
wire 20,80 u 20;
wire 5,93 u 6;
wire 10,87 u 6;
wire 15,81 u 6;
wire 30,80 u 20 r 20 d 20;
wire 30,90 r 20;
wire 60,100 d 10;
wire 65,90 d 8;
wire 70,80 u 2;
wire 75,90 d 8;
wire 80,100 d 10;
wire 90,100 d 10 r 20 u 10;
wire 100,80 u 10;
wire 130,100 d 4;
wire 140,80 u 12 r 20 d 12 l 21;
wire 142,92 u 8 r 16 d 8;
wire 170,80 r 20 u 20 l 15;
wire 190,90 l 10;
wire 200,100 d 4;

}

final.sym

1	47 A2
2	51 B1
3	55 A1
4	59 B0
5	63 A0
6	67 PHI1
7	71 PHI2
8	16 S15
9	17 S14
10	18 S13
11	19 S12
12	20 S11
13	21 S10
14	22 S9
15	842 S8
16	1468 S7
17	2955 S6
18	3403 S5
19	4262 S4
20	4875 S3
21	5443 S2
22	6507 S1
23	7349 S0
24	3 vdd
25	37 gnd
26	11331 CIN
27	11326 B15
28	11321 A15
29	11316 B14
30	11311 A14
31	11306 B13
32	11301 A13
33	11296 B12
34	11291 A12
35	11286 B11
36	11281 A11
37	11276 B10
38	11271 A10
39	11266 B9
40	10863 A9
41	10658 B8
42	10279 A8
43	8897 B7
44	8869 A7
45	7352 B6
46	6511 A6
47	5446 B5
48	4878 A5
49	4265 B4
50	3407 A4
51	2958 B3
52	1487 A3
53	845 B2

sim.in

K PHI1 011000 PHI2 000011
W A A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
W B B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
W OUT S15 S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0
W CIN CIN
h A13 A10 A9 A8 A6 A5 A4 A0 B14 B13 B9 B7 B6 B5 B1 CIN
l A15 A14 A12 A11 A7 A3 A2 A1 B15 B12 B10 B11 B8 B4 B3 B2 B0
c
l A13 A8 A4 A0 B13 B7 B5 CIN
h A1 A2 A7 A11 B2 B5 B11 B12
c
h A13 A4 A0 B7 B5 CIN
c
l A9 A8 A4 A0 B14 B9 B5 B1
c
l A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 CIN
l B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
c
h A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
c
l A15 A14
h B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0 CIN
c
c
c
c
c

sim.out

```

1  2418 transistors, 1546 nodes (1233 pulled up)
2  CIN=1 1
3  OUT=XXXXXXXXXXXXXXXXXXXXX
4  B=0110001011100010      25314
5  A=0010011101110001      10097
6  cycle took 1681 events
7  CIN=0 0
8  OUT=XXXXXXXXXXXXXXXXXXXXX
9  B=0101101001100110      23142
10 A=0000111011100110      3814
11 cycle took 1391 events
12 CIN=1 1
13 OUT=XXXXXXXXXXXXXXXXXXXXX
14 B=0101101011100110      23270
15 A=0010111011110111      12023
16 cycle took 1264 events
17 CIN=1 1
18 OUT=1000101001010100      35412
19 B=0001100011000100      6340
20 A=0010110011100110      11494
21 cycle took 1440 events
22 CIN=0 0
23 OUT=0110100101001100      26956
24 B=0000000000000000      0
25 A=0000000000000000      0
26 cycle took 1380 events
27 CIN=0 0
28 OUT=1000100111011110      35294
29 B=0000000000000000      0
30 A=1111111111111111      65535
31 cycle took 1423 events
32 CIN=1 1
33 OUT=0100010110101011      17835
34 B=0111111111111111      32767
35 A=0011111111111111      16383
36 cycle took 1583 events
37 CIN=1 1
38 OUT=0000000000000000      0
39 B=0111111111111111      32767
40 A=0011111111111111      16383
41 cycle took 1317 events
42 CIN=1 1
43 OUT=1111111111111111      65535
44 B=0111111111111111      32767
45 A=0011111111111111      16383
46 cycle took 1261 events
47 CIN=1 1
48 OUT=1011111111111111      49151
49 B=0111111111111111      32767
50 A=0011111111111111      16383
51 cycle took 989 events
52 CIN=1 1

```


53	OUT=1011111111111111	49151
54	B=0111111111111111	32767
55	A=0011111111111111	16383
56	cycle took 796 events	

LIST OF REFERENCES

1. Mead,C. and Conway,L., *Introduction To VLSI Systems*, Addison-Wesley, 1980.
2. Thomas,R.T. and Yates,J., *A User Guide To The UNIX System*, McGraw-Hill, 1982.
3. *UNIX Programmer's Manual*, Bell Laboratories, 7th ed., 1979.
4. Purdum,J., *C Programming Guide*, Que, 1983.
5. Saxe,T., *CLL - A Chip Layout Language*, version 3, paper obtained from Stanford University.
6. Wolf,W., *Design Validation For EE271*, paper obtained from Stanford University.
7. Newkirk,J., Mathews,R., Redford,J., and Burns,C., *Stanford nMOS Cell Library*, 1st ed., 1981.
8. Cohen,D. and Richardson,L., *MOSIS User's Manual*, USC/ISI, 1982.
9. Hwang,K., *Computer Arithmetic Principles, Architecture, And Design*, Wiley, 1979.
10. *Ilogs User's Manual* version 2H.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Superintendant Attn: Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
2. Dr. Robert Mathews AEL 205 Stanford, CA 94305	1
3. Lt.Col. Harold Carter AFIT/ENG Wright Patterson AFB, Ohio 45433	1
4. Lt. Joseph R. Conradi 1102 Spruance Rd. Monterey, CA 93940	2
5. Lt. Bruce R. Hauenstein 4216 Maintree Ct. Fairfax, VA 22033	1
6. Capt. Mark Stotzer 1028 Spruance Rd. Monterey, CA 93940	1
7. Mr. Albert Wong Code 52 Naval Postgraduate School Monterey, CA 93943	1
8. Ms. Susan Taylor AEL 205 Stanford, CA 94305	1
9. Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
10. Chairman, EE Department Code 62 Naval Postgraduate School Monterey, CA 93943	1
11. Dr. Donald Kirk Code 62KI Naval Postgraduate School Monterey, CA 93943	15

- | | |
|---------------------------|---|
| 12. Prof. Robert Strum | 1 |
| Code 62ST | |
| Naval Postgraduate School | |
| Monterey, CA 93943 | |
|
 | |
| 13. Dr. H.H.Loomis | 1 |
| Code 62LM | |
| Naval Postgraduate School | |
| Monterey, CA 93943 | |

206066

Thesis

C7432 Conradi

c.1

VLSI design of a very
fast pipelined carry
look ahead adder, by
Joseph Robert Conradi
and Bruce robert Hauen-
stein.

206066

Thesis

C7432 Conradi

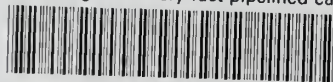
c.1

VLSI design of a very
fast pipelined carry
look ahead adder, by
Joseph Robert Conradi
and Bruce robert Hauen-
stein.



thesC7432

VLSI design of a very fast pipelined car



3 2768 002 09349 4

DUDLEY KNOX LIBRARY